# UNIVERSIDAD AUTÓNOMA DE MADRID

## ESCUELA POLITECNICA SUPERIOR



**Grado en Ingeniería de Servicios y Tecnología de Telecomunicaciones (EUR-ACE®)**

# TRABAJO FIN DE GRADO

## Circuitos Digitales Combinacionales en FPGAs

**Domingo Duque Casero**
**Tutor: Eduardo Iván Boemo Scalvinoni**

**Junio 2019**

# Circuitos Digitales Combinacionales en FPGAs

**AUTOR: Domingo Duque Casero**
**TUTOR: Eduardo I. Boemo Scalvinoni**

**Laboratorio de Sistemas Digitales**
**Dpto. Tecnología Electrónica y de Comunicaciones**
**Escuela Politécnica Superior**
**Universidad Autónoma de Madrid**
**Junio de 2019**

# Resumen (castellano)

Este Trabajo Fin de Grado tiene como objetivo el estudio de área, retardo y consumo de potencia de circuitos combinacionales sobre una FPGA.

El TFG forma parte de un proyecto conjunto entre Universidad de la República (UDELAR, Uruguay) y el DSLab UAM. La idea es crear un *benchmark suite "*UDELAR-UAM" de circuitos para medir la calidad de los resultados de estimación de potencia de herramientas EDA profesionales de Xilinx e Intel (ex Altera). El TFG colabora con un *subset* compuesto por 6 circuitos parametrizables para la operación de multiplicación y suma binaria.

El marco tecnológico se limita a la empresa Xilinx. En particular, el programa Vivado 2017.1, junto con una placa Arty A7 con una FPGA de la familia Artix-7.

En primer lugar, se ha realizado un estudio de diversos tipos de circuitos que podían implementarse en el *subset*, procurando que su estructura fuera compatible con el esquema fijo UDELAR-UAM utilizado para las medidas, con un especial interés por estructuras parametrizables.

Para la parte de sumadores, se han escogido 3 circuitos (*Ripple Carry Adder, Carry Look-Ahead* y el *IP CORE "Adder/Subtracter v12.0"*) mientras que, en los multiplicadores se han escogido otros 3 circuitos (multiplicador *Array Ripple*, multiplicador *Array Look-Ahead* y el *IP CORE "Multiplier v12.0"*)

La comprobación de resultados de las operaciones se realiza agregando al circuito un generador de secuencias aleatorias *on-chip*, de modo que todo el método funciona dentro de la FPGA, facilitando de este modo la uniformidad entre medida física de potencia y su estimación.

Para las versiones sintetizadas e implementadas, se han medido los principales aspectos para cada uno de los circuitos: el retardo (camino crítico), los recursos utilizados (LUTs) y la medición física del consumo. Se ha mantenido constante la frecuencia para cada versión con diferentes número de bits y estructura.

Por último, una vez realizado la obtención de datos, se ha analizado los resultados en dos vertientes dependiendo del tipo de operación a través de su comparación entre sí y observar las diferencias apreciadas, para concluir que tipo de circuito es el más idóneo para cada situación.

Como conclusión, se ha podido observar que el método "UDELAR-UAM" frente a circuitos que utiliza bajos recursos no resulta eficiente, como es el ejemplo de los sumadores implementados, los cuales ofrecen resultados incoherentes. Sin embargo para los multiplicadores, siendo estos, los circuitos destinados para la medición de este método, se obtienen buenos resultados. Respecto a los aspectos medibles, se ha comprobado que tienen una relación muy estrecha. La modificación de uno trae consigo la variación de los demás.

# Palabras clave (castellano)

FPGA, ASIC, consumo de potencia, retardo, área, benchmark, sumador, multiplicador, VHDL.

# Abstract (English)

The objective of this TFG is the study of the area, time delay and power consumption of combinational circuits over a FPGA.

The TFG is part of a joint project between Universidad de la República (UDELAR, Uruguay) and the DSLab UAM. The idea is to create a circuit UDELAR-UAM benchmark suite to measure the quality of the results of professionals Xilinx and Intel (ex Altera) power estimation EDA tools. This TFG collaborates with a subset composed by 6 parametrizable circuits for binary multiplication and addition operations.

The technological framework is limited to Xilinx Company. Particularly it has been used the Vivado 2017.1 program together with an Arty A7 plate with a FPGA of Artix-7 family.

First of all, a study of the different types of circuits that could be implemented in the subset has been performed, trying that its structure was compatible with the fixed UDELAR-UAMscheme used for the measurements, with a special interest for parametrizable structures.

In terms of the adders, 3 circuits have been chosen (*Ripple Carry Adder*, *Carry Look-Ahead* and the *IP Core Adder/Subtracter v12.0*). On the other hand, 3 other circuits have been chosen in terms of multipliers (*Array Ripple multiplier*, *Array Look-Ahead multiplier* and *IP CORE Multiplier v12.0*).

The operations results verification is performed adding a random sequence generation on-chipto the system, so that all the method works inside the FPGA. This combination facilitates the uniformity between physical power measurement and its estimation.

In respect of synthesized and implemented versions, the main aspects of each circuit have been added: time delay (critical way), used resources (LUTs) and the consumption physical measure. The frequency has been remained constant for each version with different number of bits and structure.

Finally, once the data were obtained, results were analysed in two different ways depending on the type of the operation through their comparison between them. The appreciated differences were observed, in order to conclude what type of circuit is the ideal for each situation.

To conclude, it has been possible to observe that UDELAR-UAM method is not efficient in circuits that used low resources. For example, implemented adders, that offers incoherent results. However for the multipliers, being these, the circuits intended the measurement of this method, good results are obtained. Regarding the measurable aspects, it has been proven that they have a very close relationship. The modification of one brings with it the variation of the others.

# Keywords (inglés)

FPGA, ASIC, power consumption, delay, area, benchmark, adder, multiplier, VHDL.

## *Agradecimientos:*

*En primer lugar, esta dedicatoria va especialmente a mis padres, José y Margarita, que a lo largo de esta etapa me recordaron insistentemente la importancia del trabajo duro de cada día y como no, de su apoyo incondicional.*

*Gracias a Eduardo Boemo, que me inspiró desde el inicio del grado al gusto referido a la rama de la Electrónica, por la enseñanza de las ideas principales de este proyecto y la facilitación del uso de las herramientas requeridas para su realización.*

*También agradecer a Federico Favaro (colaborador del grupo UDELAR) por su ayuda y su especial atención a lo largo de este proyecto.*

*Gracias a todos.*

# INDICE DE CONTENIDOS

# INDICE DE FIGURAS

# INDICE DE TABLAS

# INDICE DE FÓRMULAS

# 1 Introducción

## *1.1 Motivación*

A diferencia de los procesadores, las FPGAs son circuitos integrados compuestos por interconexiones entre bloques de lógica, con la posibilidad de ser ambos reconfigurables. Esto último supone una gran ventaja frente a los circuitos integrados denominados ASICs, los cuales una vez construidos, solo tienen esa función particular y no puede ser modificados.

Esta versatilidad da rienda suelta a la imaginación, pues permite materializar en silicio ideas a muy bajo costo. Sólo se requiere tiempo para poder implementarla. Además, gracias a Internet, existen páginas de libre acceso que dan la opción de crear código a partir de la modificación de otros diseñadores; este proyecto es un ejemplo de ello.

Actualmente, gracias a las ventajas que surgen a partir de las FPGAs, se ha permitido una mayor personalización de los sistemas electrónicos, adaptándolos a las necesidades particulares de un proyecto. La utilización de FPGAs es intensiva en diversos campos de la tecnología como son:

- Comunicaciones.
- Visión artificial.
- Prueba de diseños de bajo consumo.
- Aplicaciones miliares
- El manejo de procesamiento de imágenes o de grandes volúmenes de datos como sucede en el tema de codificación y encriptación, realizado a tiempo real.

## *1.2 Objetivos*

El objetivo de este proyecto ha sido realizar un análisis de la terna Área-Retardo-Consumo, de 6 circuitos combinacionales, agrupados en sumadores y multiplicadores.

Observando la variación que se origina a partir de la modificaciones implementadas. Quiero destacar en el aspecto del consumo de potencia, dar especial mención a la colaboración realizada con el grupo UDELAR, cuyo esquema ha sido esencial para la toma de resultados.

## *1.3 Organización de la memoria*

La memoria consta de los siguientes capítulos:

- Capítulo 1: Introducción.
- Capítulo 2: Estado del Arte.
- Capítulo 3: Benchmark
- Capítulo 4: Conceptos a desarrollar en el análisis.
- Capítulo 5: Integración y pruebas
- Capítulo 6: Resultados.
- Capítulo 7: Conclusiones y trabajo futuro.

Para facilitar una lectura fluida, a lo largo de la memoria se ha decidido utilizar los nombres de terminología extranjera en su idioma origen, a partir del formato Italic o con comillas.

# 2 Estado del arte

## *2.1 FPGA*

Es un dispositivo basado en la configuración de sus bloques de lógica para la implementación de un circuito integrado, su realización tiene un proceso similar en todos los casos, donde el punto de salida es a partir de la descripción del comportamiento mediante un lenguaje de descripción de hardware de relativamente bajo nivel (*HDL*) hasta su implementación.

## *2.2 Historia*

Esta tecnología surgió a mediados de la década de los 80, a manos de Ross Freeman, cofundador de una de las empresas que actualmente lideran esta industria, Xilinx Inc.
Freeman especuló que los transistores experimentarían una caída en el precio, llevándolo a la creación de esta idea precursora, que después, derivó en el término acuñado actual FPGA, como una opción más económica, por volumen de fabricación, a pesar de gastar más área de silicio que el equivalente *masked* ASIC.

Las FPGAs son una versión superior de las CPLD, cuya base fue combinar el tiempo de desarrollo de los PLDs, primeros dispositivos lógicos programables compuestos por puertas OR y AND, con una mayor densidad de pines y una estructura más versátil gracias a la implementación de arrays matriciales compuesto por bloques lógicos. Será por esto último, el término originario por aquel entonces como LCAs ("*Logic Cell Array"*).

En esa época, la creación de un circuito integrado con aspecto programable ya era algo cotidiano, aunque por parte de Xilinx establecieron tres fundamentos principales:

- El uso de los multiplexores para la realización de funciones lógicas, como forma de sustitución de los transistores N y P de los *gate arrays*.

- El proceso de mapeado, similar al de la telefonía analógica: conectar bloques a partir de un sistema de conmutación. De modo que se diseñó una estructura compuesta de silicio y metal que formaría el conjunto de pistas y matrices de interconexión programables por el usuario. Transistores CMOS se utilizarían para crear una interconexión y multiplexores para funciones lógicas.

- El último fundamento se derivó de las dos anteriores: crear un sistema que permitiera almacenar las configuraciones convirtiendo esta estructura en algo no volátil. Los diferentes métodos para solventar este problema fueron varios dictaminados por los diferentes fabricantes de esta tecnología: la implementación de una memoria EPROM, externa o interna, el uso de una EEPROM o el uso de antifusibles.

## *2.3 Descripción de la estructura interna de una FPGA*

Está formada por una estructura estándar similar para cualquier FPGA. Consta principalmente de bloques lógicos configurables (*CLBs*), bloques de enrutado

(comúnmente denominado como “matriz de interconexiones” o *SBs*), gestores de reloj (*DCM*) y bloques de entrada/salida (IOB) como forma de comunicación con el usuario.

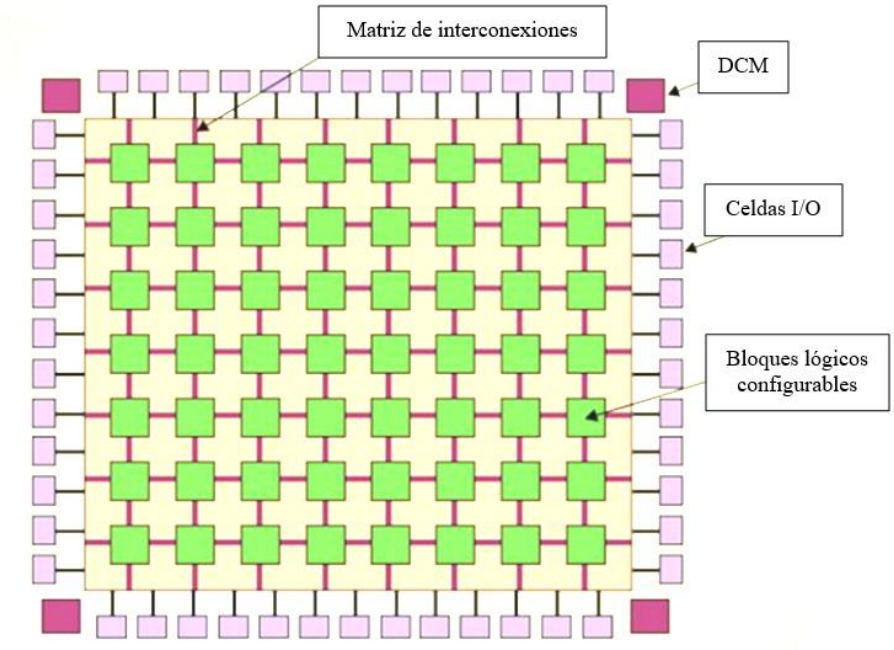



**Figura 2-1: Estructura básica de una FPGA**

Una vez comentados, procedemos a detallarlos con más profundidad:

- Los segmentos o celdas (**CLB)**, son los bloques lógicos configurables, la unidad básica de cualquier FPGA. Su función es la implementación de la lógica requerida por el usuario. Están compuestos por dos componentes principales, FFs y LUTs (“*Look up Table*”). La forma de combinarlos dependerá de la estructura de la FPGA.
  Dependiendo del número de segmentos que está compuesto, se puede clasificar según su granularidad. Granularidad fina será referido al tamaño de la menor unidad funcional que puede manipular la FPGA, esto conlleva una alta flexibilidad, pero sufre mayores retardos. En cambio, el otro tipo granularidad gruesa, los efectos producidos serán totalmente al contrario.

- Los bloques de entrada y salida (***IOB***) son los necesarios para que exista una adaptación de señal entre la parte interna de la FPGA y el mundo exterior.

- Los gestores de reloj digital (**DCM**), se encargan de proporcionar la precisión de la señal de reloj, eliminando desfases y sesgo del reloj (*clock skew*). El número que se compone en una FPGA es directamente proporcional a su tamaño.

- Las matrices de enrutado (***SB***) tienen como función establecer conexión entre bloques CLB, además de los IOB. Pueden funcionar con flujo bidireccional y operación tri-estado (establece el uso de tres niveles lógicos: alta impedancia, ‘1’ y ‘0’, dando la posibilidad de establecer entre diferentes puntos la misma salida), además de poder controlar de manera digital el conjunto de estándares de tensión e impedancia.

- Los ***multiplicadores***, componentes que en el apartado 3.2 indagaré más a fondo, como su nombre indica, tienen como función la multiplicación de dos números. Algunos de ellos vienen insertados como *cores* en el silicio, siendo una forma directa de ahorro de CLBs.

- Por último, los *cores* de ***memoria RAM***, principales responsables de almacenar datos intermedios para los procesos que los requieran.

## *2.4 Entorno de desarrollo*

El entorno de desarrollo permite la síntesis física de circuitos digitales sobre FPGAs, a partir de un diseño en lenguaje de descripción de hardware (VHDL).

En nuestro proyecto, hemos utilizado el programa *Vivado Design Suite*, con versión 2017.1. Este entorno es compatible con las siguientes familias de FPGAs: *UltraScale, Virtex-7, Kintex-7, Zynq-700* y *Artix-7*. Siendo esta última la utilizada en este trabajo.

### 2.4.1 Metodología para la implementación de un circuito

Las diferentes etapas hasta la implementación de un circuito en una FPGA son:

- **Creación del diseño** formado por VHDL, Verilog o algún lenguaje de programación para la descripción de circuitos. De modo que esto establecerá los bloques necesarios que requerirá el sistema.

- **Simulación:** Una vez revisado que el circuito escrito está libre de errores del lenguaje, comenzará el proceso de comprobación del correcto funcionamiento. En Vivado la herramienta por defecto utilizada y que en nuestro caso hemos manejado es: “Vivado Simulator”.

- **Proceso de “constraint” del circuito**: Para configurar que los componentes de la FPGA tengan el comportamiento idóneo, hará falta modificar las restricciones físicas y temporales para su síntesis. En nuestro caso, para la tarjeta utilizada será necesario descargar el archivo XDC de la página oficial de DIGILENT [1].

- **Proceso de Síntesis:** Será la etapa que convierte el lenguaje de descripción en un circuito físico generado. El término síntesis hace referencia usualmente al proceso de conversión de texto a puertas lógicas o multiplexores.

- **Proceso de Implementación:** El objetivo de esta esta etapa es la conversión del diseño sintetizado aun formato compatible con los recursos físicos específicos de la FPGA.

- **Proceso de Programación y Debug:** Sera el último paso, la realización del volcado de los valores que complementan la configuración de la FPGA (*Bitstream*), de modo que la placa se comporte de la manera que se ha previsto.

## *2.5 Principales fabricantes*

Desde la aparición de las FPGAs, ha existido en el mercado, un número pequeño de empresas debido a la inexistencia de este sector, que actualmente ha tomado impulso apareciendo cierta competitividad.

- **Xilinx**: Es la empresa en cabeza en el desarrollo y fabricación de esta tecnología, se basa en la combinación de características de FPGA, SoC y 3DICs, así como el desarrollo de entornos para el proceso de software. Posee las mayores familias de dispositivos de FPGAs: *Spartan* (actualmente retirada), *Virtex, Kintex y Artix.*

- **Intel (ex Altera)**: La otra empresa predominante en este mercado. Altera fue adquirida por Intel en 2015, se centró en el estudio de FPGAs, la combinación de SoC-FPGAs, CPLDs y ASICs, además del soporte de software. La demanda en Altera cubre cualquier gama en este mercado tecnológico. Las familias de sus productos más reconocidos son: *MAX, Strantix, Cyclone* y *HardCopy.*

- **Lattice Semiconductor:** Otro fabricante de dispositivos de alto rendimiento como FPGA, CLPD y SPLD, además de señales mixtas programables y productos de interconexión. Destacado por el lanzamiento de FPGAs con tecnología de 90nm. Sus productos principales son: *ECP* y la serie *XP* de FPGAs.

- **QuickLogic:** Fabricante de FPGAs y de ESPs, aparte de productos con sistema embebido. Tiene una tecnología basado en fusibles, programable para un único uso.

- **Otros: Actel, Atmel, Achronix Semiconductor y MathStar In.**

## *2.6 Placa utilizada*

En la realización de este proyecto se ha utilizado la tarjeta "Arty A7 35T" que tiene implementado una FPGA Artix-7 perteneciente a la familia XC7A35T, producto desarrollado por una de las empresas pioneras anteriormente nombradas, Xilinx INC.
Está orientada específicamente para sistemas de procesadores *Soft MicroBlaze*.

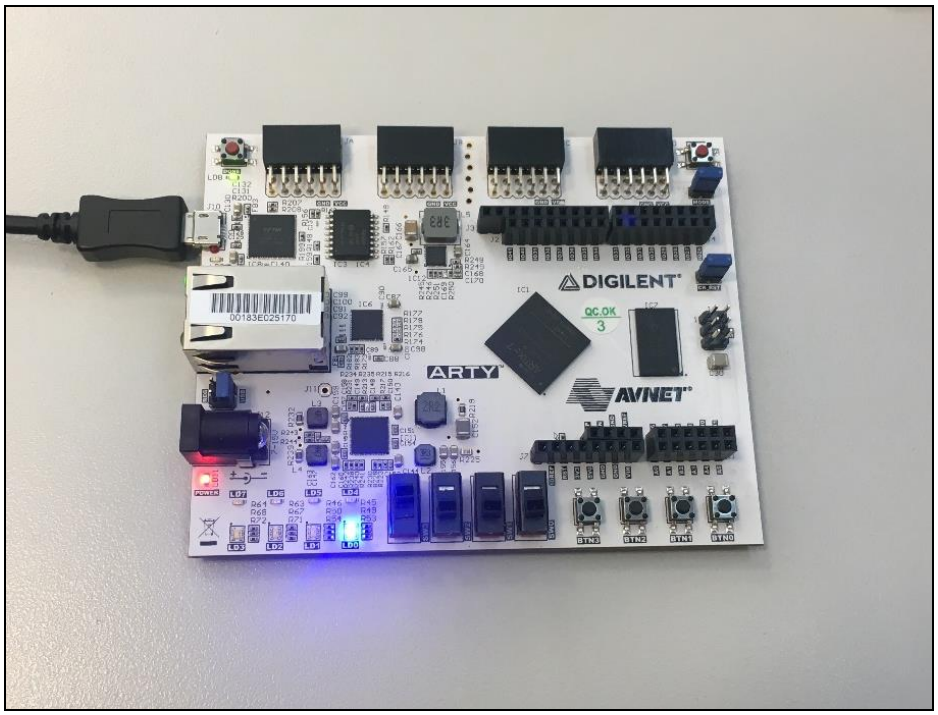



**Figura 2-2: Placa "Arty A7 35T"**

### 2.6.1 Características de la Placa

| ***Xilinx Artix-35T FPGA*** | *xc7a35ticsg324-1L* |
|---|---|
| *Celdas lógicas* | 33.280 |
| *Sectores lógicos* | 5.200 |
| *Sectores DSP* | 90 |
| *Bloque de memoria RAM (KBits)* | 1.800 |
| *Flip-Flops* | 41.600 |
| *Frecuencia máx. de funcionamiento* | 464 MHz |
| *Rango de I/O* | 210 |

**Tabla 2-1: Características de la placa "Arty A7 35T"**

# 3 Benchmark

En este apartado se va a exponer los distintos circuitos que se han elegido para la realización del análisis. Todo ello con un objetivo clave, elaborar una investigación que examine la relación que existe en términos de área, retardo y potencia, dependiendo de su estructura como en la variación del tamaño estipulado tanto en las entradas como en la salida.

Este *benchmark* incluye 6 diseños, divididos en dos clases: sumadores y multiplicadores.
Comenzaré mencionando los sumadores, ya que son el bloque básico de cualquier operación, incluido los multiplicadores.

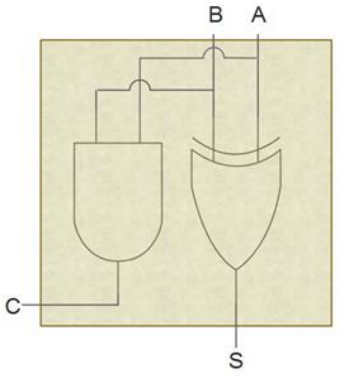
## *3.1 Sumadores*

Los sumadores son los componentes con mayor importancia de las operaciones aritméticas básicas en el diseño de circuitos, se encuentra en cualquier tipo de procesador a través de la ALU para el cálculo de direcciones de memoria, índice de tablas y operaciones similares.

Su objetivo principal es realizar la suma de dos números binarios, la mecánica utilizada será similar al método común ejercido para los números decimales con diferencia del rango que supone esta lógica, ceros y unos.
Para el caso en que tratamos con números decimales y realizamos esta operación, si el resultado obtenido es un número mayor a 9 determinamos un valor de acarreo, que se deberá sumar en la siguiente columna del dígito de mayor peso. Por tanto, realizado para las normas de la suma binaria, en este caso a partir de la suma de 1+1, aparecerá como resultado (*S*) '0' y como acarreo (*C*) '1'.

Para agilizar estas operaciones, se crean bloques que permitan instanciar dicha operación, apareciendo los Sumadores (*Full-Adder*) y los Semisumadores (*Half-Adder*), donde este último difiere del primero por la inexistencia de una entrada de acarreo inicial.

- **Semi-sumador (*Half- Adder*)**



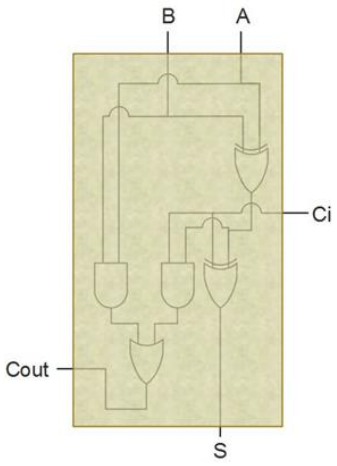

$$S = A \oplus B$$

$$C = A \circ B$$

| A | B | C | S |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

**Figura 3-1: *Half-Adder*, E/S y tabla de verdad**

- **Sumador (*Full-Adder*)**




$$Cout = AB + AC_i + BC_i$$
$$S = A \oplus B \oplus C_i$$

| A | B | Ci | Cout | S |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

**Figura 3-2: *Full-Adder*, E/S y tabla de verdad**

### 3.1.1 Sumador *Ripple Carry* (*RCA*)

A partir del anterior bloque "Full-Adder", surge el método *Ripple*, cuyo procedimiento es la implementación de un número N de bloques "Full-Adder", dependiendo del tamaño del vector de entrada de los dos números en cuestión.
De modo que cada "Full-Adder" contenga el bit de la misma posición de cada palabra, incluyendo además el acarreo final de la anterior suma parcial de los dos bits de menor posición.

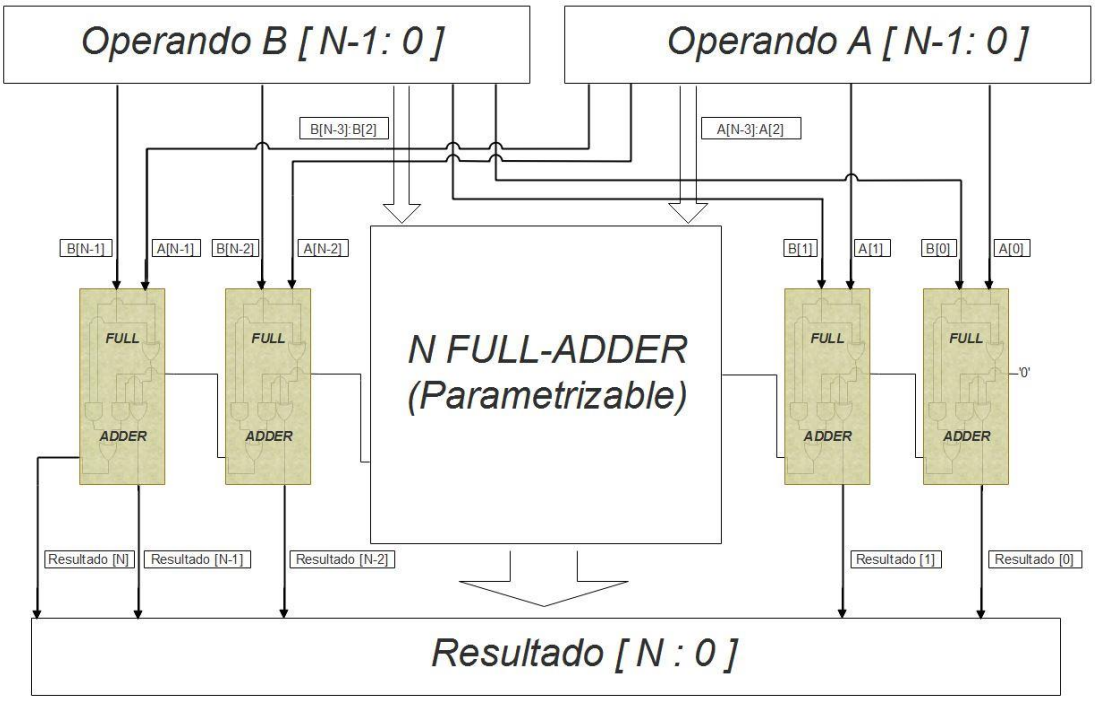



**Figura 3-3: Método *Ripple Carry***

Como resultado, se obtiene la suma de ambos bits y su acarreo, dicho valor será el suministrado en la siguiente suma parcial.

Esto último presenta un impedimento, ya que no se realiza una suma en paralelo, es necesario realizar la espera de la transmisión del acarreo en todas sus etapas para que se efectúe correctamente la operación.

### 3.1.2 Sumador *Carry Look-Ahead* (*CLA*)

Este sumador, denominado "sumador paralelo con acarreo anticipado", corrige el impedimento que sufre el método Ripple, calculando de antemano los bits de acarreo.

El procedimiento utilizado será la adaptación de un nuevo bloque, donde a partir de dos señales, "Gi" y "Pi", deducen paralelamente los valores del Carry inicial de todas las etapas en el mismo momento.

$$P_i = A_i \oplus B_i$$

$$G_i = A_i * B_i$$

**Fórmula 3-1: Señales G y P, Sumador "Look-Ahead"**

A continuación, se muestran las fórmulas tanto de la salida como del acarreo generado a partir de las dos nuevas señales:

$$S = P_i \oplus C_i$$

$$C(i+1) = G_i + P_i * Ci$$

**Fórmula 3-2: Salidas en función de las nuevas señales, Sumador "Look-Ahead"**

Las ecuaciones de generación de acarreo para las distintas etapas será la combinación del acarreo originado en la etapa anterior, por tanto, todas las ecuaciones están relacionadas entre sí. Para facilitar el entendimiento dejo a continuación, la ecuación obtenida para el acarreo final para 4 bits

$$C_1 = G_0 + C_0 * P_0$$

$$C_2 = G_1 + C_1 * P_1 = G_1 + P_1 * (G_0 + C_0 * P_0) = G_1 + G_0 * P_1 + C_0 * P_0 * P_1$$

$$C_3 = G_2 + C_2 * P_2 = G_2 + G_1 * P_2 + G_0 * P_1 * P_2 + C_0 * P_0 * P_1 * P_2$$

$$C_4 = G_3 + C_3 * P_3 = G_3 + G_2 * P_3 + G_1 * P_2 * P_3 + G_0 * P_1 * P_2 * P_3 + G_0 * P_0 * P_1 * P_2 * P_3$$

**Fórmula 3-3: Cálculo de la serie de acarreo, Sumador "Look-Ahead"**

Si queremos implementar un esquemático, dividiremos el "Full-Adder" en dos bloques independientes, que trabajen al margen uno del otro, de tal manera que realice el objetivo propuesto.

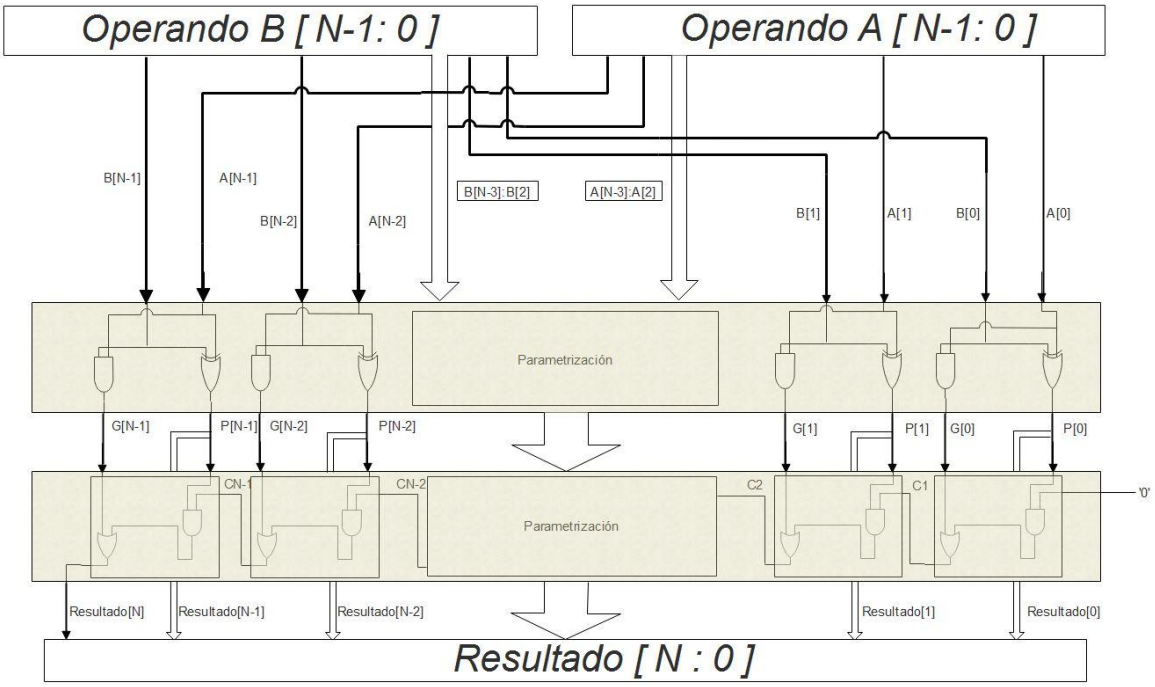



**Figura 3-4: Método *Carry Look-Ahead***

### 3.1.3 Adder / Subtracter (IP Catalog - Vivado)

Una de las características que nos permite los lenguajes de descripción, es la posibilidad de reutilización del código creado de proyectos previos, de manera que existen modelos de diferentes bloques ya testeados. Por tanto, a través de una fácil configuración podemos estructurarlos de la manera que necesitemos.

Hasta ahora hemos trabajado con el término "componente", a partir de un código visible, se realiza la modificación de sus características, creando nuestro bloque. El siguiente caso no se formará de la misma manera, esta vez será a través de los *IP Cores*, las FPGAs como he ido comentando a lo largo de este documento, permite la incorporación de diseños complejos.
Estos bloques de lógica están organizados en paquetes para poder instanciarse directamente en nuestros proyectos, de modo que su modificación es más limitada, permite a partir de un menú simple editar las propiedades y características de su estructura. Además de adjuntarnos una pequeña información para conocer su funcionamiento. Podemos observar el total de sus entradas y salidas a través del esquemático que nos muestra, de modo que podemos modificar su tamaño, en caso en que sea posible.

En nuestro entorno Vivado, está opción también ha sido implementada. Existe un apartado denominado "IP Catalog", donde a partir de una biblioteca de recursos, podremos buscar el bloque en cuestión, además de poder examinar un sinfín de diseños de diferentes tipos como CPUs, ALUs, bloques de aritmética etc.

En nuestro caso se obtendrá de la siguiente manera:

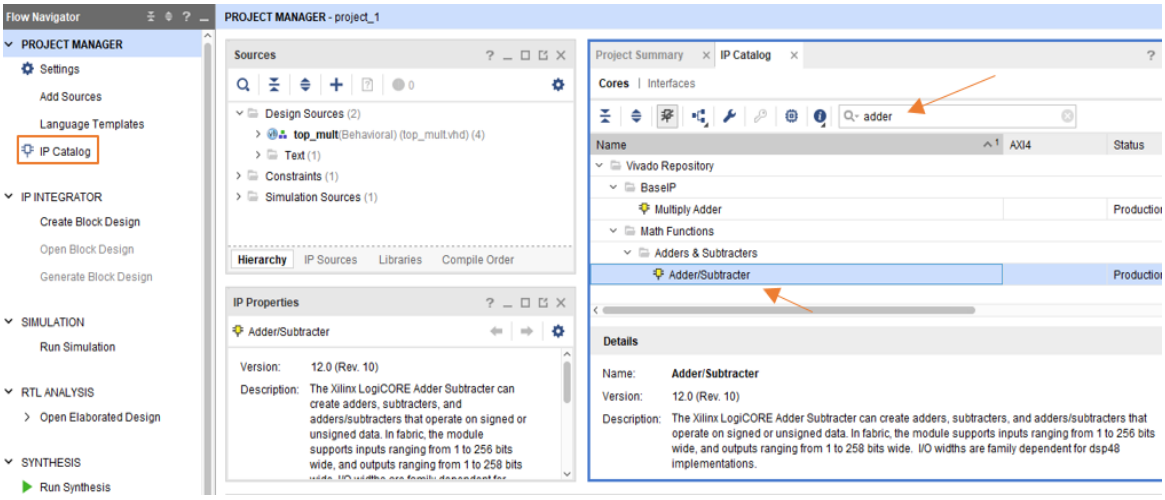



**Figura 3-5: Localización del *Adder / Subtracter***

- **Adder/Subtracter v12.0**

Se trata de un componente que puede operar tanto la suma como la resta a partir de números con o sin bit de signo. En nuestro caso, mi objetivo es la operación de la suma, selecciono la configuración "add" y números sin bit de signo.

La descripción de los puertos es la siguiente:

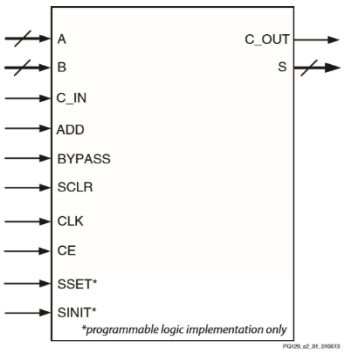



**Figura 3-6: Bloque *Adder / Subtracter v12.0***

| Name | Direction | Description |
|---|---|---|
| A[N:0] | Input | A Input bus |
| B[M:0](1) | Input | B Input bus |
| ADD | Input | Controls the operation performed by an Adder/Subtracter (High = Addition, Low = Subtraction) |
| C_IN | Input | Carry Input |
| C_OUT | Output | Carry Output |
| S[P:0] | Output | Output bus |
| BYPASS | Input | Bypass control signal loads B port onto S port |
| CE | Input | Active-High Clock Enable |
| CLK | Input | Clock signal: rising edge |
| SCLR | Input | Synchronous Clear: forces outputs to a Low state when driven High |
| SINIT(2) | Input | Synchronous Initialization - forces outputs to a user defined state when driven High |
| SSET(2) | Input | Synchronous Set - forces outputs to a High state when driven High |

**Tabla 3-1: Tabla de entradas y salidas, *Adder / Substracter v12.0***

Su fabricación puede ser de dos maneras, a partir de la implementación del bloque computacional complejo "DSP48" configurado a tres tipos de estructura, las cuales dependerán de la situación de los registros. La segunda opción es la implementación de fábrica ("Fabric"), dando la opción de configurar su pipeline modificando el número de secciones de división del bus. Esta forma te permite almacenar menos datos, obteniendo una mayor eficiencia, a partir de la creación de un diseño más pequeño. Hemos optado por escoger la segunda opción.

Los siguientes apartados, son referidos a su estructura, activando o desactivando las entradas de control y modificando el tamaño de bits. Respecto a la suma, existen dos ajustes para configurarlo, realizando la operación cuyos factores vienen destinados por dos entradas, o la suma continua a través de la utilización de una única entrada y un valor determinado. En nuestro caso, hemos seleccionado la primera opción.

La última configuración (y necesaria para este trabajo) es el aspecto de retardo ("Latency") relacionado con la estructura de pipeline. Como este trabajo se basa en el uso de circuitos con sistema combinacional, no existe esa señal de reloj, de modo que habrá que configurarlo para un valor '0'.

La configuración queda de la siguiente manera:

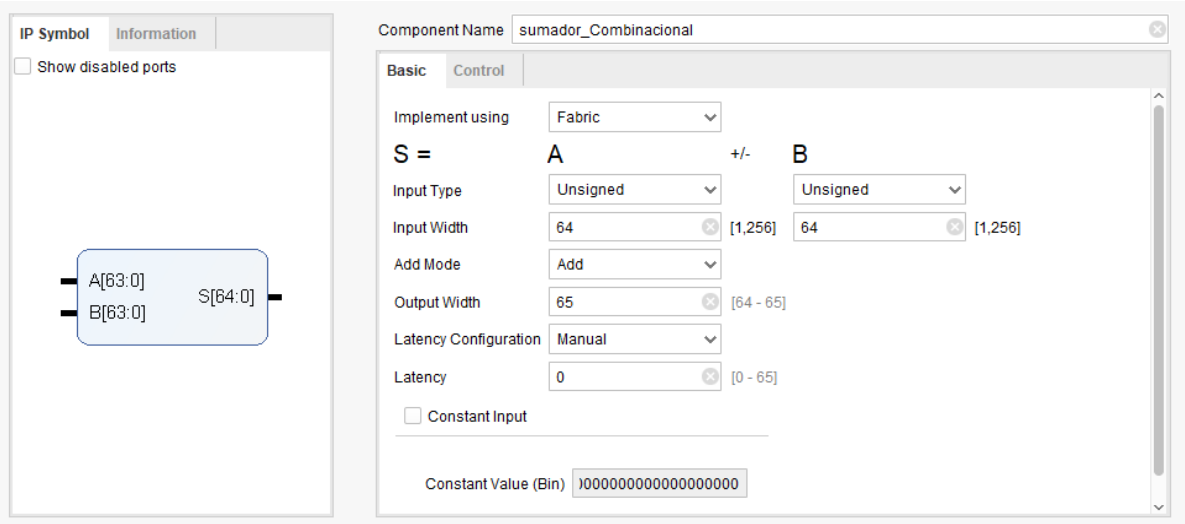



**Figura 3-7: Configuración *Adder / Subtracter v.12***

Una vez realizado, obtendremos el bloque nuevo con el nombre proporcionado. El último paso es su instancia en el proyecto.

La documentación de este bloque IP Core se encuentra en la página web de Xilinx [3]

## *3.2 Multiplicadores*

La multiplicación es la segunda operación más utilizada, por detrás de la suma. Es un circuito capaz de realizar la operación multiplicar de dos números (N y M), obteniendo su producto (P). En el mundo de la tecnología es uno de los bloques funcionales principales de cualquier sistema digital, viene integrado en cualquier procesador, causante de sincronizar todos los componentes y operaciones en un intervalo de tiempo. Por ello, con el paso de los años ha sufrido una gran evolución con relación a su demanda.

El procedimiento realizado tiene como objetivo la obtención de la matriz parcial del producto. Donde cada valor se obtiene a partir de la puerta lógica AND, necesitando por tanto un número de N*M de estas puertas para un multiplicador N x M.
El siguiente paso es la reducción del número de filas de los productos creados a través de la suma, de tal manera que la estructura se va simplificando hasta la obtención del resultado.

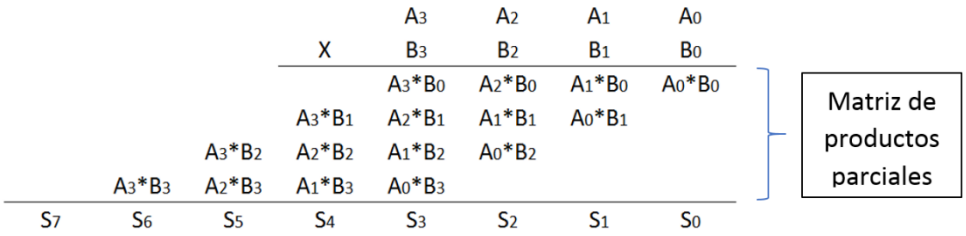



**Figura 3-8: Multiplicación, Matriz de productos parciales**

Según el célebre físico Chris Wallace, autor de uno de los circuitos más conocidos "Wallace Tree Multiplier", determinó en su artículo "*A suggestion for a Fast Multiplier*" las tres pautas principales para conseguir la mejor eficiencia, todo relacionado con el término de productos parciales, es decir, el producto de cada par de bits del multiplicador. Estas pautas son:

- Disminución del tiempo en su generación
- Disminución del conjunto en el que se compone
- Aumento en la velocidad de la suma.

Conocido esto, se presentan tres multiplicadores, cuya complejidad depende exponencialmente del número de bits que se componen las entradas. Cada circuito elabora un procedimiento, en el cual, únicamente se diferencia de la metodología utilizada para la agrupación y suma de los valores de la matriz parcial, por lo demás la idea es similar.

### 3.2.1 Multiplicador Array Ripple

El multiplicador Array Ripple implementado, se trata de un método que aplica el conocido "Algoritmo de desplazamiento y acumulación". Teniendo a la vista la imagen de la matriz de productos parciales, el procedimiento es el siguiente:

Comienza con la obtencion de la primera fila como resultado, tras la realización de la operación AND, teniendo como operandos el primer dígito del multiplicador y la sucesión de bits del multiplicando. Una vez hecho, se realiza el desplazamiento de una posición,

siempre de derecha a izquierda (del menor bit significativo al mayor significativo) para colocar la siguiente fila. Esta fila se trata de forma similar, pero esta vez tomando el segundo bit del multiplicador. El proceso se realiza sucesivamente hasta ejecutar todos los bits del multiplicador. Obteniendo N filas, siendo N el número de bits que contiene el multiplicador. Una vez terminado, el resultado es obtenido tras la ejecución de la suma de todas las filas en el orden correcto.

Para implementar este procedimiento en un componente, se ha tenido especial interés en el trato realizado al bit del multiplicador, efectuando las siguientes operaciones:

- Si el dígito del multiplicador (B) es igual a '0', la fila resultante será una fila contenida de ceros, asemejando la realización de la operación AND para cada dígito teniendo como el otro factor el valor '0'.

- Si el digito es '1', esa fila es igual al vector del multiplicando, realizando la misma idea anterior.

Una vez sabido esto, el método comienza con la formación del primer número de la suma acumulativa, tomando el primer bit del multiplicador. Si observamos con detenimiento la Figura 3-8, especialmente en la estructura final del resultado (S), es necesario que, para cada vector obtenido de cada suma, debido a su estado volátil, se sustraiga el contenido para no perderlo, ya que este equivale a un dígito del resultado final.

Como enuncia su nombre, este método realiza el proceso de sumas a partir de una serie de sumadores *Ripple*, necesitando M-1, siendo M el número de filas de la matriz parcial. Este decremento de un valor es debido a que, el primer resultado de la suma acumulativa se toma previamente y es utilizada como uno de los factores de la operación.

La forma ideal para implementarlo se realiza a partir de un nuevo bloque generado a partir de la estructura utilizada del *Ripple*, donde a parte de la estructura básica del sumador, contiene un multiplexor, responsable de la comprobación del valor del dígito del multiplicador. Este bloque mantiene el mismo número de entradas (A, B y Ci) y de salidas (S y Cout). Sin embargo, la forma de actuar será distinta.

Si tenemos en cuenta que no existe ningún tipo de acarreo inicial para cada fila, utilizamos esta entrada "Ci" como el bit de selección del multiplexor. Para las demás entradas, "A" y "B", contienen tanto la suma acumulativa como el multiplicando respectivamente.
Respecto a las salidas, "Cout" formará parte del valor obtenido de la suma acumulativa, insertado en la posición del bit más significativo, dicha suma es obtenida a partir de la salida "S".

A continuación, muestro un esquema de la metodología:

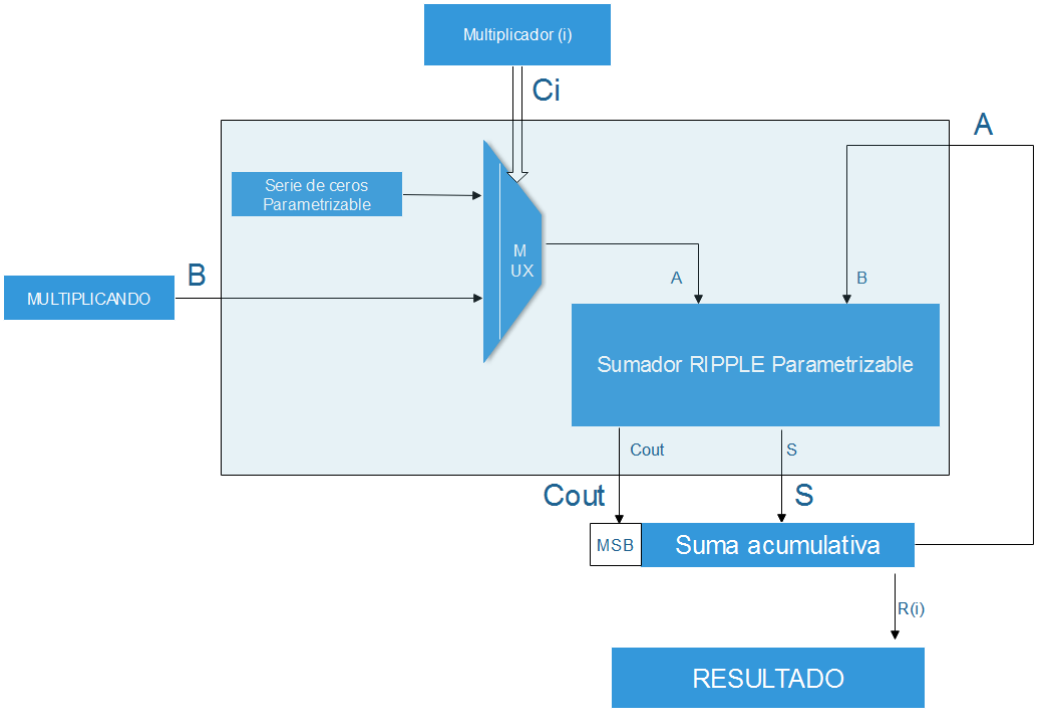



**Figura 3-9: Método Multiplicador "Array Ripple"**

### 3.2.2 Multiplicador Array LookAhead

El multiplicador "Array Look-Ahead", funciona de forma idéntica al anterior multiplicador, siendo la única diferencia el uso de sumadores "Carry Look-Ahead".

En este caso, el esquemático es el siguiente:

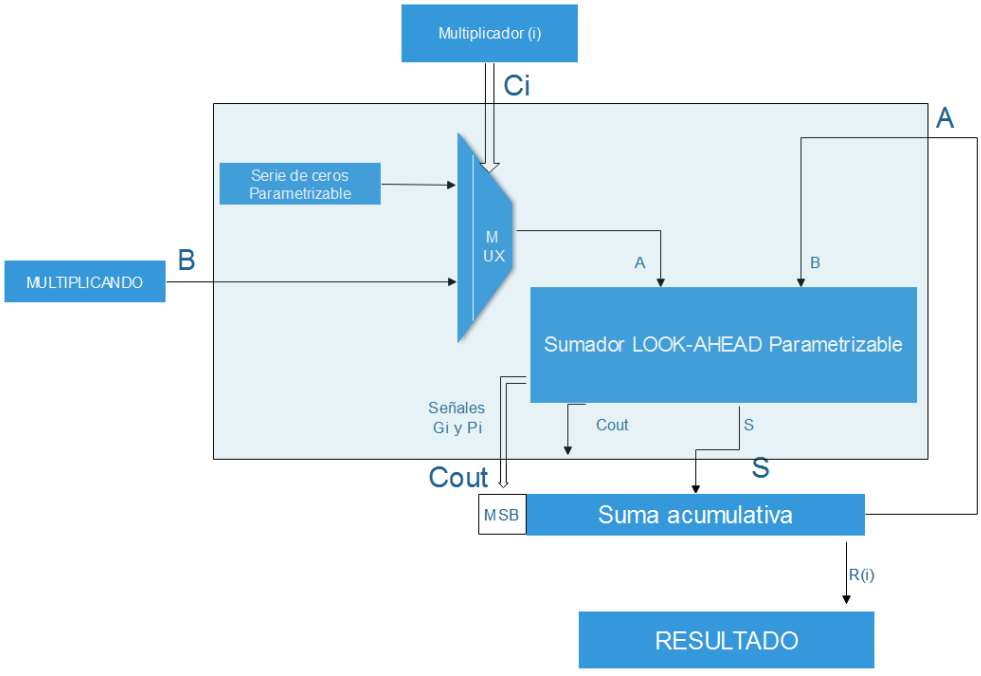



**Figura 3-10: Método Multiplicador "Array Look-Ahead"**

### 3.2.3 Multiplier (IP Catalog - Vivado)

Igual que accedíamos en el catálogo de IP CORE de Vivado para seleccionar un sumador en el Apartado 3.1.3 (Figura 3-5). En este caso, lo hacemos de forma similar, ya que existen bloques pre-diseñados (*cores*) que realizan esta operación. De modo que realizando una búsqueda, hemos seleccionado el siguiente bloque:

- Multiplier v12.0

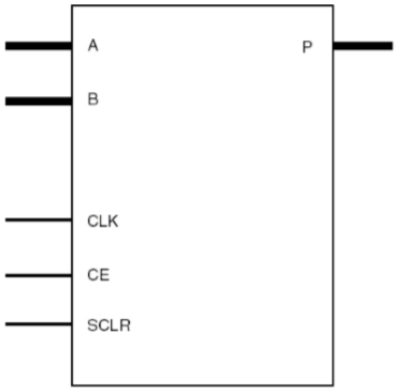



**Figura 3-11: Bloque *Multiplier* v12.0**

| Signal | Direction | Description |
|---|---|---|
| A[N-1:0] | Input | A operand input bus, N bits wide |
| B[M-1:0] | Input | B operand input bus, M bits wide (parallel multipliers only) |
| CLK | Input | Rising-edge clock input |
| CE | Input | Active-High Clock Enable |
| SCLR | Input | Active-High Synchronous Clear (SCLR/CE priority is configurable) |
| P[X:Y] | Output | Product Output – bit X down to bit Y |

**Tabla 3-2: Tabla de entradas y salidas, Multiplier v12.0**

Este multiplicador puede ser configurado a partir de dos tipos: "Multiplicador paralelo" para el uso de dos entradas como factores para realizar la operación o "Multiplicador de coeficiente constante" en el que implementa uno de los operandos con un valor constante.

Respecto a las opciones de entrada, podemos variar su tamaño y trabajar con o sin bit de signo. En nuestro caso, optaremos por esta última.

La siguiente opción viene referido al tipo de implementación que tendrá su núcleo, tenemos dos opciones, la fabricación a partir de multiplicadores (DSP) o de LUTS. Tomaremos la segunda opción debido al método utilizado para la estimación del área.

La siguiente tabla de selección, deberemos de indicar el objetivo de preferencia que queremos ejecutar, el ahorro de tamaño o la máxima velocidad, en este caso nos hemos decantado por la opción por defecto "Speed Optimized".

Por último, como estamos tratando con sistemas combinacionales, tomaremos la misma opción que en el apartado de sumadores, realizando la operación en ausencia de una señal de reloj. Por tanto, situaremos en el apartado de "Pipeline Stages" el valor '0'.

# 4 Desarrollo

El objetivo propuesto de este trabajo es el análisis de los distintos circuitos, comparando sus características respecto a los conceptos de retardo, área y consumo de potencia.

## 4.1 Retardo

El retardo es el tiempo que tarda un circuito en realizar una operación. Comienza cuando se efectúa la transición de la entrada y acaba cuando se refleja en la salida.

La operación de circuito digital requiere actualmente varios nanosegundos. Este retardo está relacionado con los parámetros distribuidos del circuito y la tecnología de conmutación de los propios transistores. El retardo depende de la temperatura y la tensión.

### 4.1.1 Conceptos de Retardo

El retardo o latencia de un circuito (recordando que todos los circuitos son combinacionales en este TFG) es determinado por aquel camino, el cual origina el mayor retardo ("camino crítico" o *critical path*).

He de destacar también el concepto ***Slack***, será a partir de él, con el que obtendremos los diferentes resultados. Su definición es la diferencia o margen entre el tiempo máximo de llegada y el efectivo, necesario para conocer la capacidad de ejecución en términos de velocidad del que derivará el retardo.

La frecuencia máxima, relacionado con el término citado anteriormente, es la frecuencia que establece la velocidad del componente, la fórmula es la siguiente:

$$f_{clk\ max} = \frac{1}{t_{ce} + t_{setUp} + t_{fTranMax}}$$

**Fórmula 4-1: Frecuencia máxima**

Siendo $t_{ce}$ el tiempo utilizado para realizar el cambio de estado (flanco de subida), $t_{setUp}$ será el tiempo que necesitan las salidas para que se pueda registrar el dato correctamente y $t_{fTranMax}$ será el tiempo de propagación del bloque más lento, lo cual será el principal responsable de la velocidad del circuito.

## 4.2 Área

El tamaño de un circuito es la porción de silicio de la FPGA necesaria para la creación de un circuito. Normalmente viene relacionada con el número de celdas utilizadas (LUTs).
El entorno Vivado incorpora estos datos además de su porcentaje, en los que se muestra la relación que existe entre las celdas totales y las celdas que han sido utilizadas por el circuito en cuestión (sección de *"Project Summary").*

Respecto a los bloques surgidos del catálogo de *IP CORE,* existe una estimación del área utilizada, proporcionando el número de elementos básicos que requiere el bloque.

## 4.3 Consumo de potencia

Desde ya hace varias décadas el error relativo de los estimadores de potencia en FPGA es grande. Es decir, hay una gran diferencia entre la potencia hallada a partir de un modelo teórico y la potencia real (a partir de herramientas de medición). Se suele evaluar con la siguiente fórmula:

$$Error = \frac{Potencia\ estimada - Potencia\ medida}{Potencia\ medida}$$

**Fórmula 4-2: Error relativo de potencia**

Es decir, las herramientas de estimación de potencia ("Xpower" y "PowerPlay") en el mundo de las FPGAs producen resultados erróneos. A lo largo de estos años, diversos grupos de investigación han realizado comparaciones que evidencian lo anterior. La creación de un *benchmark suite* trata de unificar este tipo de experimentos.

### 4.3.1 Conceptos de consumo de potencia

El consumo de potencia es la disipación de energía por unidad de tiempo de un circuito electrónico. El método para calcularla es usualmente medir la corriente que entra al circuito:

$$P = I * V$$

**Fórmula 4-3: Potencia**

La potencia, además, tiene diferentes términos:

- La **potencia dinámica,** será aquella potencia consumida cuando el circuito en cuestión está en ejecución, es decir, realizando la operación determinada. Es uno de los mayores problemas que se originan en la creación de circuitos de altas frecuencias de trabajo. Viene determinado por la acción de carga y descarga de los condensadores de un circuito, de modo que su ecuación es la siguiente:

  $$Pot_{din} = C_{eff} * V^2 * f_{clock}$$

  **Fórmula 4-4: Potencia dinámica**

  Siendo V la tensión del alimentador, $f_{clock}$ la frecuencia en la que opera el circuito y $C_{eff}$ (capacidad efectiva) relación de la capacidad de cada nodo y el factor de actividad.

- La **Potencia estática,** también denominada como de reposo, será el consumo efectuado cuando el dispositivo está únicamente encendido.

  $$P_{est} = I_{est} * V$$

  **Fórmula 4-5: Potencia estática**

# 5 Integración y pruebas

Una vez presentados los aspectos que se van a estudiar, comienza el análisis que dependiendo del parámetro establecido, se realizará de una determinada manera.

Para los aspectos de retardo y área, serán calculados a partir del entorno Vivado desde el "proceso de Implementación". Mientras que el aspecto de consumo de potencia se realiza a partir del esquema elaborado por el conjunto UDELAR-UAM, un método a nivel de hardware; es decir, la obtención de la toma de resultados es directamente sobre la FPGA.

He de comentar que las medidas realizadas han sido a partir de opciones "por defecto", sin variar ninguna de las opciones.

## *5.1 Estimación del retardo*

En el aspecto de retardo, como he comentado anteriormente, será necesario conocer la frecuencia máxima en la que operan los distintos circuitos, para después clasificarlos dependiendo de su velocidad de ejecución.

En este caso, Vivado no nos presenta directamente el valor explícito de la frecuencia máxima del circuito, pero se podrá calcular a partir del término *Worst Negative Slack* (WNS) obtenido a partir de la opción de "Report Timming", en la sección de "Design Timing Summary" o la sección "Intra-Clock Paths".

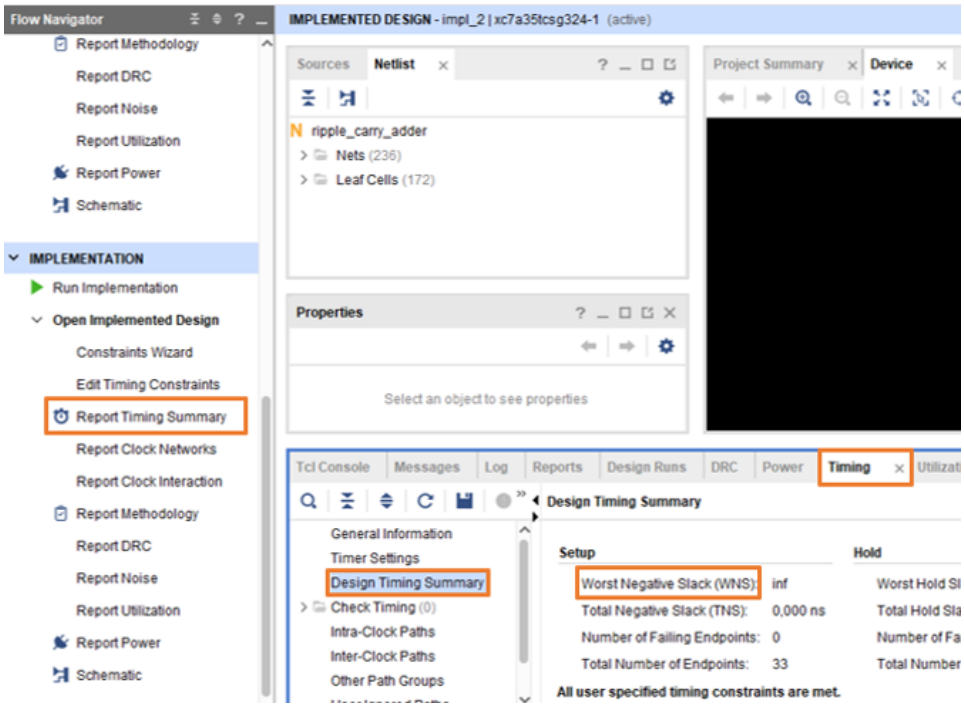



**Figura 5-1: Obtención del valor de WNS (*slack*)**

Sin embargo, este entorno confunde el significado de este término "WNS", ya que, por definición, si obtenemos un *slack* menor a cero el camino crítico falla, ya que la frecuencia

que está funcionando es más alta que la máxima. Sin embargo, el entorno no lo observa así, Vivado presenta este valor con el signo opuesto.

En nuestro proyecto, trabajamos sobre una frecuencia de reloj de: 100 MHz, la cual es disminuida a 25 MHz gracias a un PLL. Sabiendo que la frecuencia de reloj permanece constante para todos los circuitos, aunque su tamaño varíe. Podemos formar la ecuación de la frecuencia máxima del circuito en cuestión como:

$$Fmáx = \frac{1}{\frac{1}{25MHz} - WNS\ (ns)}$$

**Fórmula 5-1: Frecuencia máxima a partir del WNS**

Conociendo la frecuencia máxima, el retardo máximo será su inverso.

## *5.2 Estimación del Área*

Para el área, actuaremos de una forma similar, esta vez tendremos que tomar en cuenta la opción de "*Report Utilization"*, en la sección "*Summary"*.

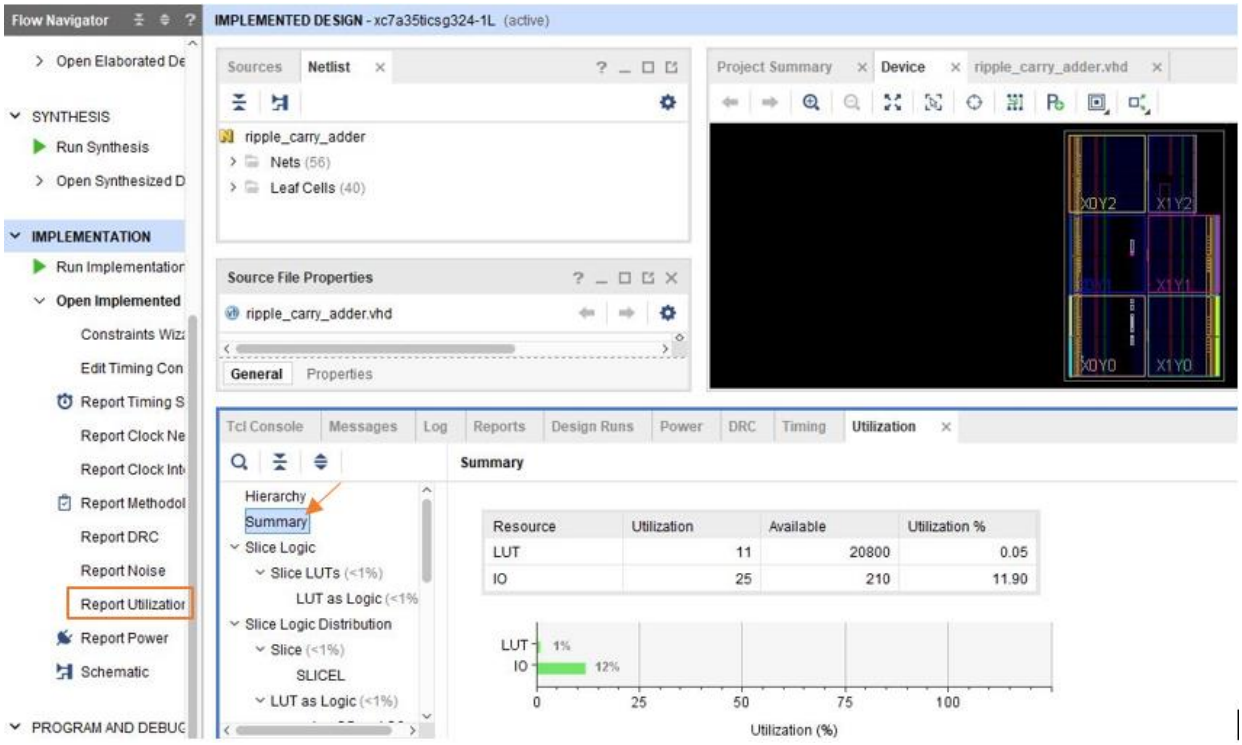



**Figura 5-2: Obtención del área**

Apareciendo los siguientes términos:

- LUTs: El número de "Look-Up table", asemejado al tamaño que se accede.
- IOB: Es el número de bits de entradas y salida de cada circuito. En el caso de los sumadores, está compuesto por 2*N bits debido a las dos entradas y N+1 bits por la

salida. En cambio, para los multiplicadores es 2*N bits por las entradas y 2*N bits por la salida.

Por tanto, para conocer el área se ha determinado a partir del número de LUTs que se han generado en el circuito implementado.

## 5.3 Estimación del consumo de Potencia

Para conocer el consumo de potencia de los diversos circuitos, se ha utilizado el esquema "UDELAR-UAM". A continuación, se expone dicho método y las diferentes partes que engloban este proceso hasta la obtención de resultados.

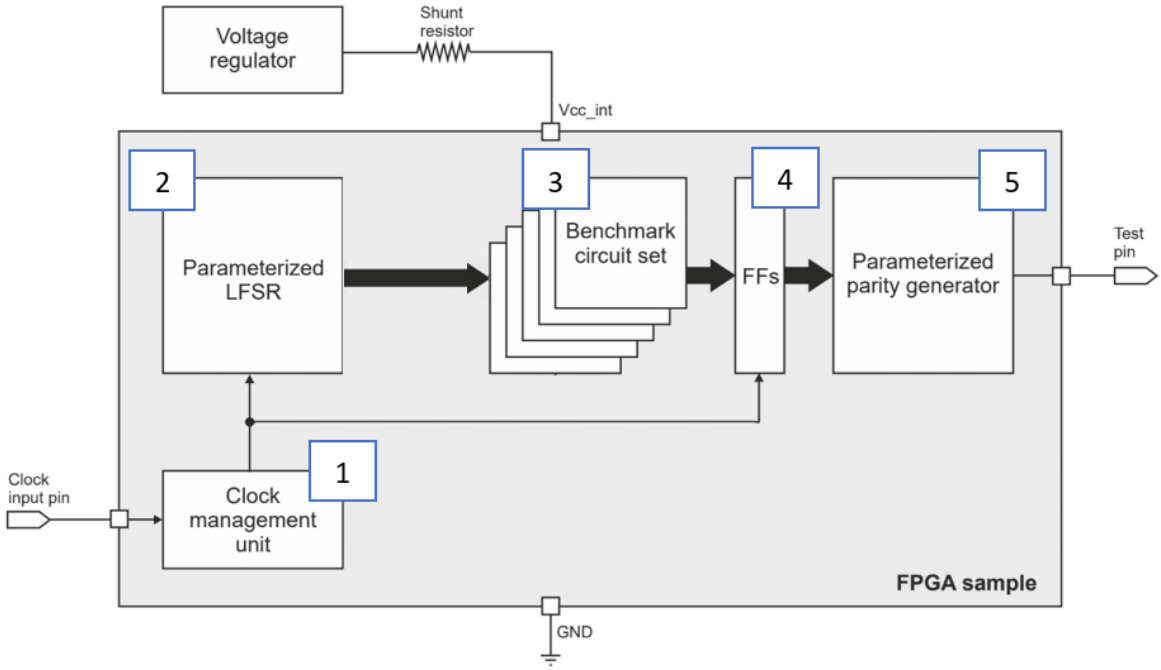



**Figura 5-3: Esquema del método "UDELAR-UAM"**

1. **Bloque de gestión de reloj**: Unidad que secuencia la E/S de datos entre los bloques.

2. **LFSR:** Registro de desplazamiento de retroalimentación lineal (véase Anexo A).

3. **Benchmark suministrado:** Será el conjunto de circuitos (en este TFG, sumadores y multiplicadores), previamente comentados en el apartado de "Benchmark".

4. **FF:** Bloques de *Flip Flops*, necesarios para que haya sincronismo con las siguientes etapas.

5. **Bloque de generador de paridad parametrizable** (Anexo B).

El circuito usa un PLL que divide una cuarta parte la frecuencia estándar (100MHz), de tal manera que el circuito funciona a una frecuencia relativamente baja, adecuada para todos los circuitos que componen el *benchmark*.

En principio, el LFSR está limitado para datos de 8 bits, pero puede parametrizarse para otros anchos. A la salida de los circuitos se han incorporado FF para sincronizar los bits del resultado.

Por último, se realiza una etapa rudimentaria de comprobación para validar los valores obtenidos. Un generador de paridad genera una firma de salida, que depende de los datos y del correcto funcionamiento del circuito. Se puede comprobar con un osciloscopio y comparar con la simulación.

Es importante comentar que, para calcular los datos en este aspecto, trabajamos a partir de un bloque compuesto por muchos componentes, donde se genera un consumo extra. Este consumo extra no debemos de tenerle cuenta en nuestro análisis.

Por tanto, para resolver este problema, aparte de obtener los resultados de todos los circuitos con los diferentes tamaños, hemos tomado las medidas para un "circuito nulo", los resultados obtenidos a partir de él equipararían a ese consumo extra.

Para realizarlo, tan solo necesitaremos emparejar la señal generada del LFSR directamente a los FF saltándonos el paso de la incorporación del benchmark.

Una vez hecho, únicamente realizando una simple resta del conjunto total, incorporando los circuitos del benchmark menos la medida obtenida del consumo extra, lograríamos la estimación del consumo que se ha adquirido para cada caso.

### 5.3.1 Obtención indirecta del valor de la corriente

El valor se calcula a través de la caída de tensión en una resistencia (*Shunt*), cuyo valor es 10 mΩ con una tolerancia de 1%. Configurando el canal 10 del XADC como una entrada, y haciendo las debidas conversiones podemos obtener el valor digital.

Adjunto una imagen donde se aprecia la estructura interna, la cual se encuentra la resistencia en cuestión. (Para más información, agrego el Anexo C que presenta esta implementación en el esquema de la Arty A7)

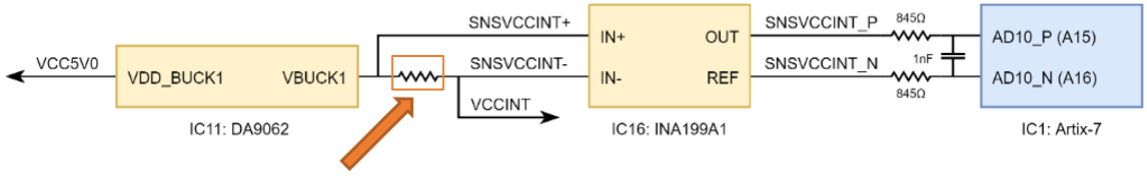



**Figura 5-4: Esquema resistencia *shunt***

El método "UDELAR-UAM" suministra un *script* con una lista de comandos para facilitarnos la obtención de la corriente. De esta manera automatizaremos el proceso de medida. Se mide la caída de tensión tomando 100 muestras por segundo, visualizándolas a partir de un monitor implementado. Una vez obtenidas las medidas, se realiza su valor medio. El proceso para la ejecución del comando se resume en el Anexo D.

Se tiene que hacer una aclaración sobre esta parte, la cual hay que tener especial atención en las unidades obtenidas, debido a que el integrado "INA 199A1" amplifica la caída de tensión obtenida con una ganancia de 50V/V.

Una vez tomado la tensión media, el siguiente paso es obtener el valor de la corriente, aplicando la Ley de Ohm y tomando en cuenta el apunte comentado anteriormente.
La fórmula obtenida es la siguiente:

$$I_{FPGA} = \frac{Vmed}{R * Gain} = \frac{Vmed}{0.01 * 50}$$

**Fórmula 5-2: Intensidad de la FPGA**

Una vez conocida la intensidad, operaríamos en términos de potencia a partir de la ecuación formulada anteriormente (Fórmula 4-3).

Conociendo el Voltaje que suministra la FPGA (core), muestro en la siguiente imagen la tensión utilizada.

| Supply | Circuits | Device | Maximum Current |
|---|---|---|---|
| 5.0V | Onboard Regulators, RGB LEDs | IC13: ON Semiconductor NCP380 | 1.0A (5.0A)[1)] |
| 3.3V | FPGA I/O, Clocks, Flash, PMODs, LEDs, Buttons, Switches, USB port, Ethernet | IC11: Dialog Semiconductor DA9062 | 2.0A |
| 0.95V (1.0V)[2)] | FPGA Core and Block RAM | IC11: Dialog Semiconductor DA9062 | 2.5A |
| 1.8V | FPGA Auxiliary | IC11: Dialog Semiconductor DA9062 | 1.5A |
| 1.35V | DDR3L and associated FPGA bank | IC11: Dialog Semiconductor DA9062 | 2.5A |
| 0.675V | DDR3L | IC17: Diodes Incorporated AP2303 | 1.75A |
| 1.25V | XADC Analog Reference | IC14: Texas Instruments REF3012 | 25mA |

**Tabla 5-1: Tabla de tensiones de la placa "Arty A7"**

# 6 Resultados

En este apartado se muestran los resultados obtenidos a partir de las medidas argumentadas anteriormente. Debido al aspecto parametrizable, la obtención de datos puede ser muy extensa. Sin embargo, debido a la limitación que ofrece la placa utilizada (Arty A7) conteniendo un número de celdas E/S determinado (210). La integración de circuitos que requieran un número mayor de estas celdas implica un error en su implementación.

Debido a esta complicación, el intervalo de dimensiones escogido respecto a la operación suma ha sido de 8, 16, 32 y 64 bits. Mientras que en la operación multiplicación se ha tomado para 8, 16 y 32 bits, descartando la opción de 64 bits por la necesidad de utilizar 256 celdas, cifra no soportada.

## 6.1 Resultados del retardo

Para el cálculo de medidas en este aspecto, su procedimiento se ha realizado sobre la estructura del método "UDELAR-UAM". De modo que los resultados vienen en consecuencia del tiempo invertido en el cómputo global.

En este apartado, se exponen tanto las frecuencias máximas como los retardos máximos para los dos tipos de circuito.

- Sumadores

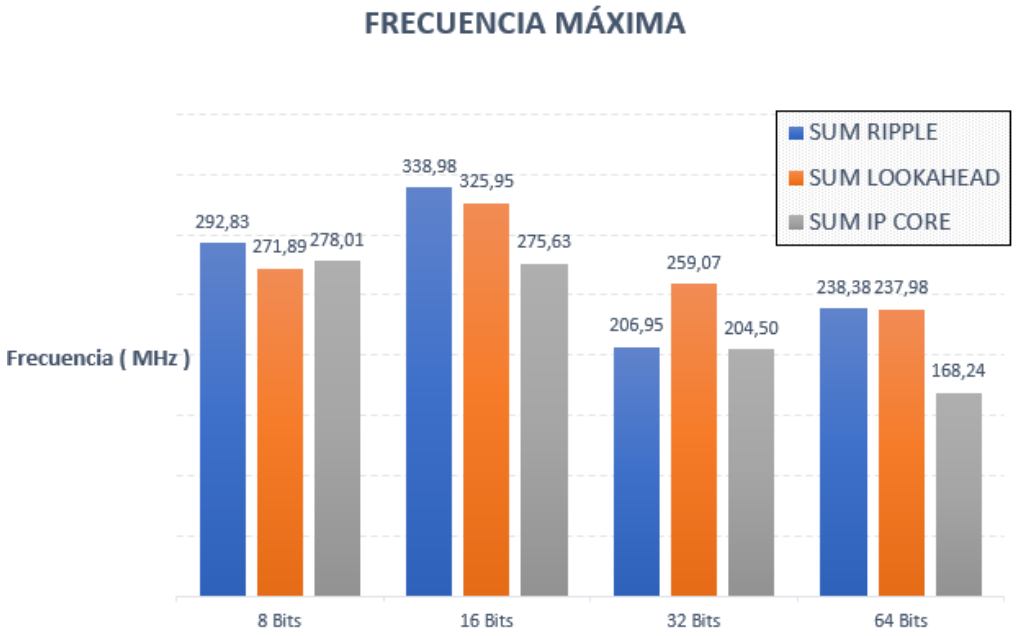



**Figura 6-1: Gráfico frecuencia máxima, Sumadores**

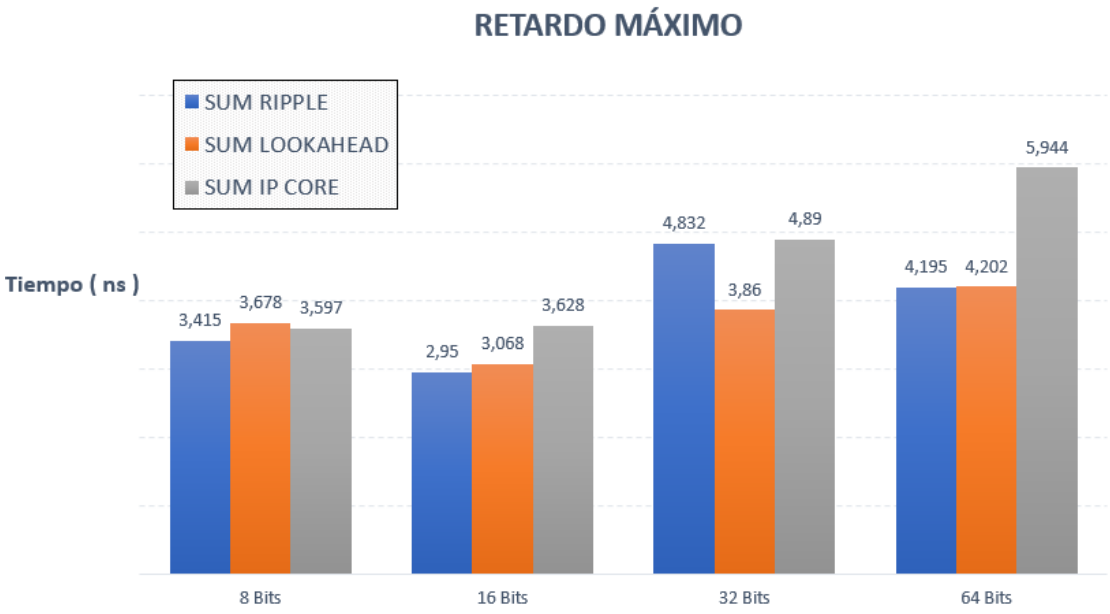



**Figura 6-2: Gráfico retardo máximo, Sumadores**

Se puede observar que para el aspecto de "timming", el uso de sumadores a partir de componentes surgidos de código visible, realizan las operaciones en mejores tiempos que el sumador desarrollado por Xilinx. También se puede observar el crecimiento del retardo generado a medida que aumenta el número de bits.

- Multiplicadores

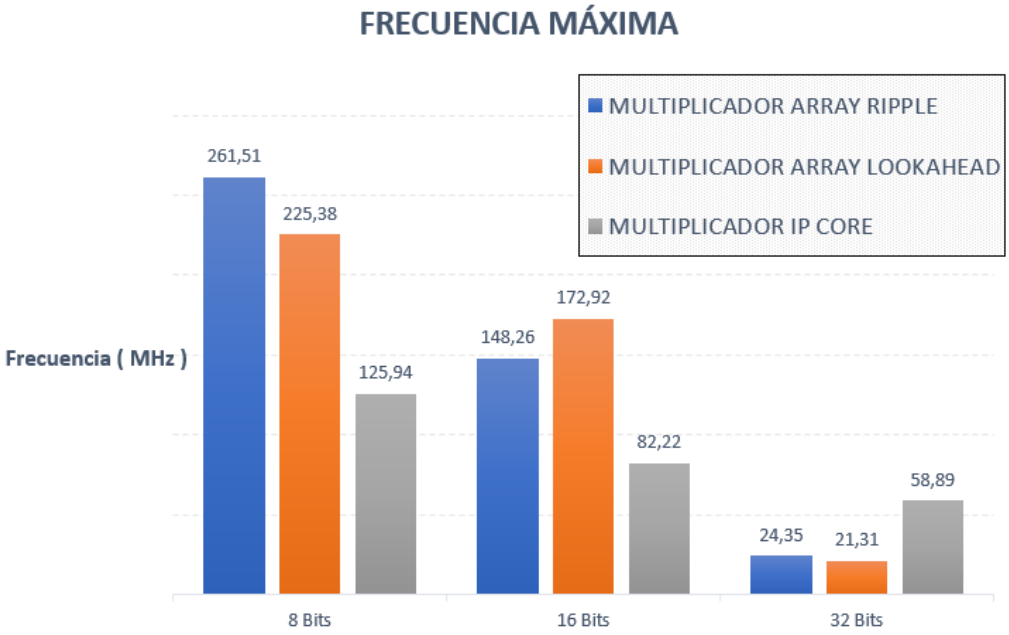



**Figura 6-3: Gráfico frecuencia máxima, Multiplicadores**

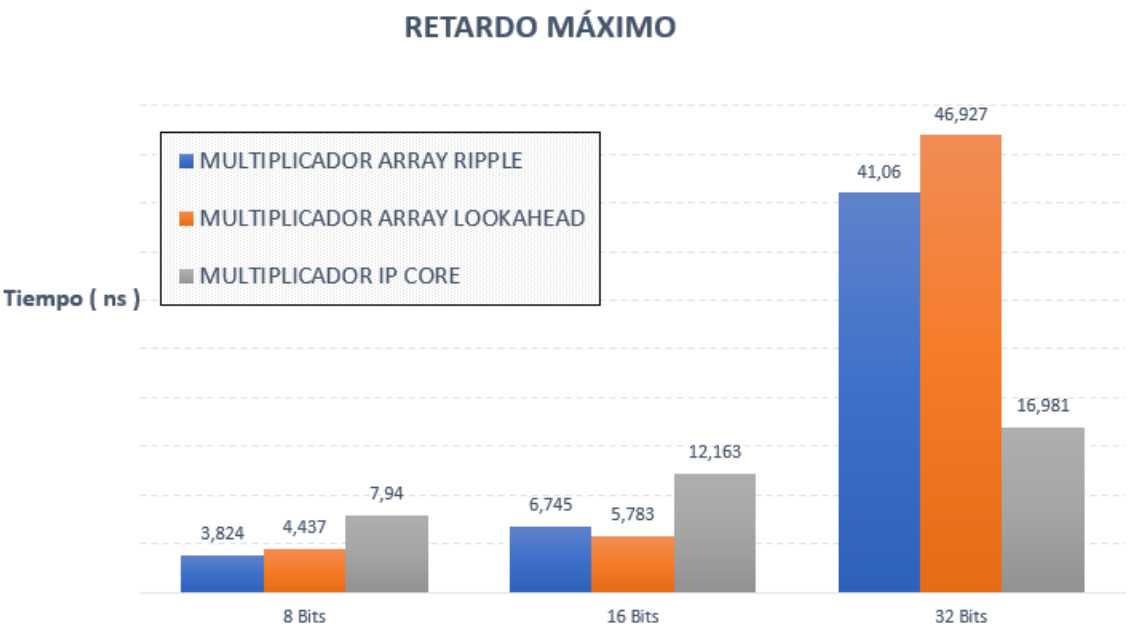



**Figura 6-4: Gráfico de retardo máximo, Multiplicadores**

Respecto a los multiplicadores, surge un impedimento tanto el multiplicador "Array Ripple" como el multiplicador "Array Look-Ahead" han traspasado el retardo máximo de funcionamiento del circuito, establecido en 40 ns (debido a la frecuencia obtenida del PLL, 25 MHz). Por tanto, los requerimientos en el aspecto "Timming" no se cumplen.

Por consiguiente, se ha realizado un barrido de frecuencia disminuyendo el rango para conocer la frecuencia máxima permitida, para que ambos circuitos de tamaño 32 bits funcionen adecuadamente. Llegando a un frecuencia máxima de 20 MHz (cuyo retardo máximo es de 50 ns), obteniendo unos tiempos de propagación de 49,834 y 49,083 ns respectivamente. Cumpliendo ambos tiempos las exigencias del circuito.

Podemos concluir, si tomamos nota del apartado de área, que debido al alto número de recursos que utilizan los multiplicadores generados para un número de bits grandes, no son una eleccion eficiente. Sin embargo, el multiplicador IP CORE mantiene un crecimiento lineal.

## 6.2 Resultados del área

En esta sección, se exponen los recursos utilizados por la FPGA para sintetizar los circuitos en el proceso de "implementación".

- Sumadores

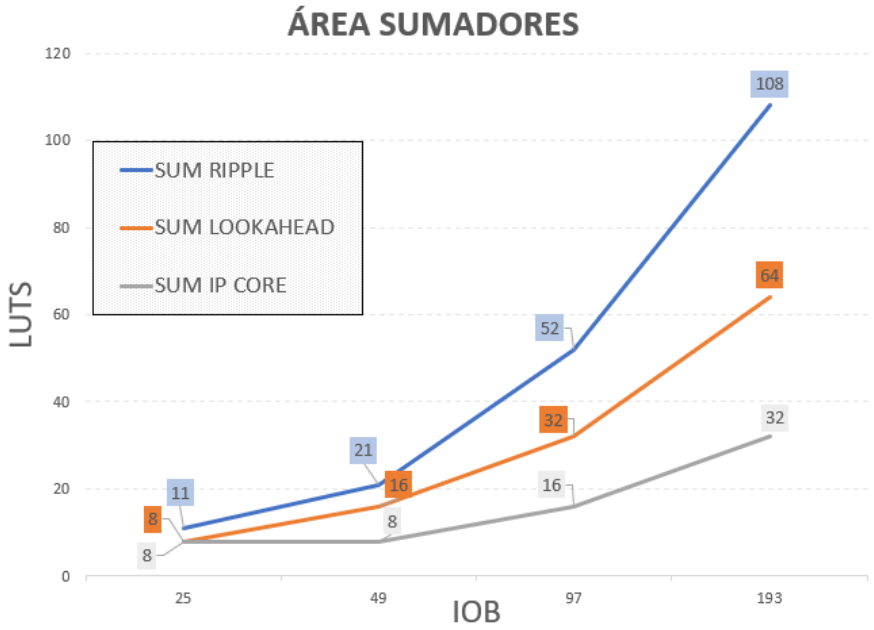



**Figura 6-5: Gráfico del área, Sumadores**

Se puede observar que el mayor tamaño alcanzado es el sumador “Ripple”, mientras que el sumador “Look-AHead” debido al procedimiento que realiza para el cálculo del acarreo, se ahorra casi un 60% en el área. Por último, se puede observar que el sumador proveniente del catálogo garantiza los mejores resultados en este aspecto.

- Multiplicadores

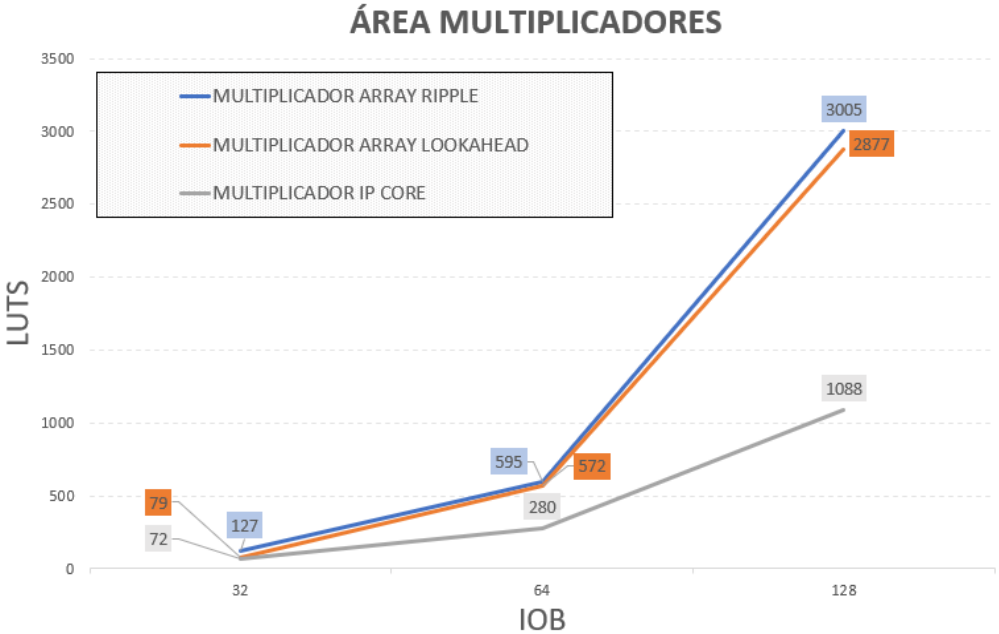



**Figura 6-6: Gráfico del área, Multiplicadores**

Respecto a los multiplicadores originados a partir de un array, las técnicas empleadas para su formación son muy similares. Es necesario un número similar de Sumadores Ripple que

de Sumadores “Look-Ahead”, debido a la misma generación del número de instancias de “Full-Adders”.
De forma similar a lo que sucede en los circuitos sumadores, el componente extraído del catálogo es el más eficiente.

### 6.3 Resultados del consumo de Potencia

Respecto a este apartado, se ha realizado una estimación del consumo para los diferentes circuitos. Sin embargo, para aquellos circuitos combinacionales con pocos recursos, como son el ejemplo de los sumadores (como hemos podido comprobar en el apartado de área). Se han obtenido resultados incoherentes, los valores extraídos de las herramientas de medida no presentan una relación medible.

Debido a esto, se ha querido descartar la tabla de resultados.

- Multiplicadores

En la siguiente tabla se puede observar los diferentes resultados obtenidos. Se ha incluido, además, la estimación del consumo para las situaciones extras, en las cuales se ha necesitado bajar la frecuencia establecida para un funcionamiento adecuado.

| | **Multiplicador Array Ripple** | | **Multiplicador Array LookAhead** | | **Multiplicador IP CORE** | |
|---|---|---|---|---|---|---|
| Nº Bits | *Intensidad (mA)* | *Potencia (mW)* | *Intensidad (mA)* | *Potencia (mW)* | *Intensidad (mA)* | *Potencia (mW)* |
| 8 x 8 | 0,1 | 0.095 | 0,516 | 0,4902 | 0,217 | 0,20615 |
| 16 x 16 | 0,384 | 0,3648 | 0,781 | 0,74195 | 0,494 | 0,4693 |
| 32 x 32 ( 25 MHz) | 68,582 | 65,1529 | 56,741 | 53,90395 | 5,361 | 5,0995 |
| 32 x 32 ( 20 MHz) | 34,909 | 33,16355 | 34,615 | 32088425 | | |

**Tabla 6-1: Estimación del consumo de Potencia**

Se puede apreciar un crecimiento exponencial del consumo con el aumento del número de bits. También podemos analizar que, si obtenemos un consumo desproporcionado, puede ser una prueba clave para conocer que el funcionamiento que realiza es incorrecto.

Vemos como, el multiplicador IP CORE es la mejor opción, el cual, no realiza un consumo tan excesivo como se presentan los otros multiplicadores.

# 7 Conclusiones y trabajo futuro

## *7.1 Conclusiones*

Este Trabajo de Fin de Grado ha tenido como función el análisis de diferentes aspectos de diseño para diversos circuitos. De tal manera, que dependiendo del objetivo planteado tomemos una elección distinta.

Para aspectos de retardo, se ha comprobado que para operaciones con un tamaño menor de bits, la mejor opción es el uso de componentes generados. Sin embargo, para situaciones en las que se operan con números de alta longitud, el aumento del retardo es exponencial, escogiendo de manera más eficiente el uso de bloques desarrollados.
Para el tema de área la elección es sencilla, los IP CORE no desaprovechan recursos para la ejecución de la operación, siendo por tanto la mejor elección.
Por último, sobre el aspecto del consumo, se ha concluido que el método de medidas utilizado "UDELAR-UAM" requería de circuitos de altos recursos. La realización de dichas pruebas en sumadores no era creíble, existía una gran imprecisión en sus cálculos. Una posible solución hubiera sido la implementación de pipeline, originando un mayor uso en los recursos. Sin embargo, esta solución requiere el uso de una señal de reloj, algo no estipulado en los sistemas combinacionales. Si hablamos de los resultados respecto a los multiplicadores, se ha podido apreciar que los IP CORE son más eficientes en este aspecto.

Si intentamos resumir los aspectos anteriormente analizados, evaluamos que los 3 aspectos estudiados van cogidos de la mano. La modificación de un parámetro trae consigo la alteración de los demás. Para una situación en que se ha planteado el aumento en el tamaños de sus entradas, ha ocasionado el incremento del área utilizado, y por consiguiente, el aumento tanto del retardo como del consumo.

## *7.2 Trabajo futuro*

Este proyecto no tiene un final, existen multitud de opciones para complementar. Tan solo con la integración de otros ejemplos de circuitos, o la elección de otros tipo de operaciones. Todo está en manos del siguiente que recoja el relevo.

Existe la posibilidad de realizar el mismo procedimiento a partir de otra FPGA con mejores prestaciones, dando la oportunidad de aumentar las capacidades bajo prueba de estudio. Este proyecto se ha realizado para dimensiones estándar, pero existía la posibilidad de la utilización de tamaños intermedios, consiguiendo por tanto una infinidad de datos.

Existen otros métodos para las estimaciones de medida, esta vez ha sido a partir un procedimiento en concreto, destinado por una colaboración. Pero existen diversos métodos y programas para la obtención de los resultados.

También, se ha contemplado para el futuro la creación de una página de web como repositorio de un benchmark destinado a la UAM, como lugar de recogida para la integración de más componentes con sus datos de evaluación. Formándose un referente como seña reivindicativa para nuestra universidad.

# Referencias


[1] https://reference.digilentinc.com/reference/programmable-logic/arty-a7/schematic
[2] IP Core: Adder / Substracter v12.0: https://www.xilinx.com/support/documentation/ip_documentation/addsub/v12_0/pg120-c-addsub.pdf
[3] IP Core: Multiplier v12.0: https://www.xilinx.com/support/documentation/ip_documentation/mult_gen/v12_0/pg108-mult-gen.pdf
[4] System Monitor: https://www.xilinx.com/support/documentation/ip_documentation/xadc_wiz/v3_0/pg091-xadc-wiz.pdf
[5] Digital Clock Manager (DCMs): https://www.xilinx.com/support/documentation/application_notes/xapp462.pdf
[6] Generador de paridad (DCMs): https://ocw.ehu.eus/pluginfile.php/2700/mod_page/content/1/Tema_5/5_11.pdf
[7] http://www.iiis.org/CDs2013/CD2013SCI/CISCI_2013/PapersPdf/CA622WJ.pdf
[8] Jean-Pierre Deschamps, Gery J.A. Bioul, Gustavo D. Sutter "Synthesis of Arithmetic Circuits_ FPGA, ASIC and Embedded Systems". Cambridge University Press
[9] Yu-Tsang_Carven Chang "Advance HDL Design Training On Xilinx FPGA-Associate Researcher"
[10] Ian Kuon, R. T. "FPGA Architecture". (2008).
[11] Jean-Pierre Deschamps, Gery J.A. Bioul, Gustavo D. Sutter "Synthesis of Arithmetic Circuits_ FPGA, ASIC and Embedded Systems". (2006)
[12] J.P. Oliver, E. Boemo: "Power Estimations ss. Power Measurements in Cyclone III Devices" (2011)
[13] Michael Keating, "Low Power Methodology Manual". (2008)
[14] Oliver,J. P. "Técnicas de Bajo consumo en FPGA". (2014)
[15] Gustavo Sutter, "Aportes a la Reducción de consumo en FPGAs (2005)
[16] https://cs.uns.edu.ar/~pmd/ac_ing/downloads/Slides/ACI-Clase-5.pdf
[17] http://www.iiis.org/CDs2013/CD2013SCI/CISCI_2013/PapersPdf/CA622WJ.pdf
FUENTES DE RECURSOS:
[18] https://www.nandland.com/vhdl/modules/module-ripple-carry-adder.html
[19] https://www.nandland.com/vhdl/modules/module-full-adder.html
[20] https://www.nandland.com/vhdl/modules/carry-lookahead-adder-vhdl.html
[21] https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2019_1/ug939-vivado-designing-with-ip-tutorial.pdf
[22] https://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug903-vivado-using-constraints.pdf
[23] https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841912/ARTY+Power+Demo
[24] http://www.andraka.com/multipli.php
[25] https://www.tamps.cinvestav.mx/~mmorales/documents/FPGAsyReconfig.pdf
[26] http://www.ijcset.com/docs/IJCSET11-02-02-01.pdf
[27] Gustavo E. Ordóñez-Fernández, Lewin A. López-López, Jaime Velasco-Medina. "Diseño de multiplicadores paralelos de 16 bits en FPGA"
[28] http://atlas.physics.arizona.edu/~kjohns/downloads/vhdl/VHDL_Lang.pdf

# Glosario

| | |
|---|---|
| UAM | Universidad Autónoma de Madrid |
| UAM | Universidad de la República (Uruguay) |
| EPS | Escuela Politécnica Superior |
| FPGA | Field Programmable Gate Array |
| LCA | Logic Cell Array |
| EPROM | Erasable Programmable Read-Only Memory |
| EEPROM | Electrically Erasable Programmable Read-Only Memory |
| PLD | Programmable Logic Device |
| SPLD | Simple Programmable Logic Device |
| CPLD | Complex Programmable Logic Device |
| SoC | System on a Chip |
| WNS | Worst Negative Slack |
| HDL | Hardware Description Language |
| FF | Flip-Flop |
| LFSR | Linear Feedback Shift Register |

# Anexos

## A LFSR: Linear Feedback Shift Register

Traducido al español como “registro de desplazamiento con retroalimentación”
Se trata de una herramienta muy utilizada en muchos campos tecnológicos como son la criptografía, en aplicaciones para comunicaciones y en hardware, este último es nuestro caso. Surgió del principio de las sucesiones recurrentes sobre los campos finitos en la lógica binaria.

LSFR es un registro de desplazamiento, cuya entrada viene transferido por una secuencia de bits del valor global combinada por la operación XOR en algunos bits.

Su estructura está formada por un conjunto de celdas y de conexiones de retroalimentación. En las celdas se sitúan todos los bits de una secuencia, mientras que las conexiones son situadas en los bits cuyas celdas van a ser combinadas. Se puede definir en la fórmula de un polinomio (conocido como “polinomio de conexiones de LFSR”). Este polinomio, como coeficientes solo podrá tener 0’s y 1’s debido a la lógica binaria.

Un ejemplo de polinomio para 16 bits tiene la forma siguiente:

$$P(x) = 1 + C_1x^1 + C_2x^2 + C_3x^3 + C_4x^4 + \cdots + C_{16}x^{16}$$

De modo que dependiendo del número que sea el coeficiente $C_n$, nos indica si el registro al que pertenece dicho coeficiente interviene en la ecuación. Siendo ‘1’ si ese registro si influye o ‘0’ para el caso contrario.

A continuación, pongo un ejemplo de la representación gráfica de un LFSR para 16 bits, cuyo polinomio es: $P(x) = 1 + x^4 + x^{13} + x^{15} + x^{16}$

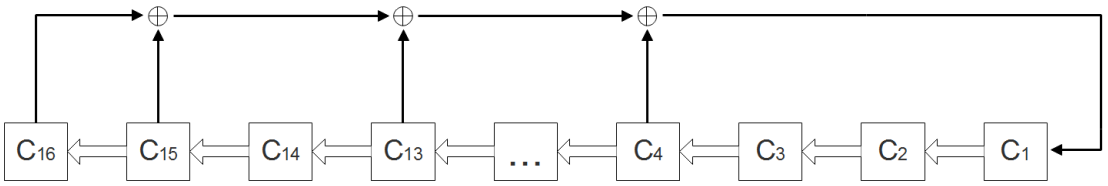



**Figura 0-1: Estructura LFSR**

Por tanto, tomando los valores de las celdas del polinomio, y haciendo la operación XOR en el orden que se observa en el dibujo (del final al principio), obtendríamos la nueva entrada resultante.
El proceso es ir sacando los valores, una vez tomados resolver las operaciones XOR que lo forman para obtener la nueva entrada. Por último, desplazamos de todas las celdas el valor contenido a la siguiente celda.

Ej. Para una secuencia inicial de 0100…1011, obtendremos 100X … 0111 (‘X’ porque no conocemos el dato)

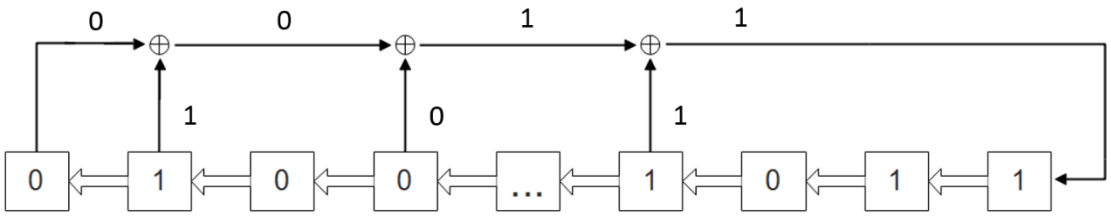



**Figura 0-2: Ejemplo de funcionamiento, LFSR**

La sucesión de estas secuencias se realizará un número de veces antes de volver a obtener el mismo estado inicial que comenzó el proceso. Este periodo de la secuencia de salida es el número de iteraciones antes del volver al estado inicial.

$$N_{veces} = 2^N - 1$$

Siendo N el número de bits que está compuesto la secuencia.

Para finalizar, dejo una captura de una simulación del LFSR para 32 bits, donde se puede observar el desplazamiento que he comentado anteriormente.

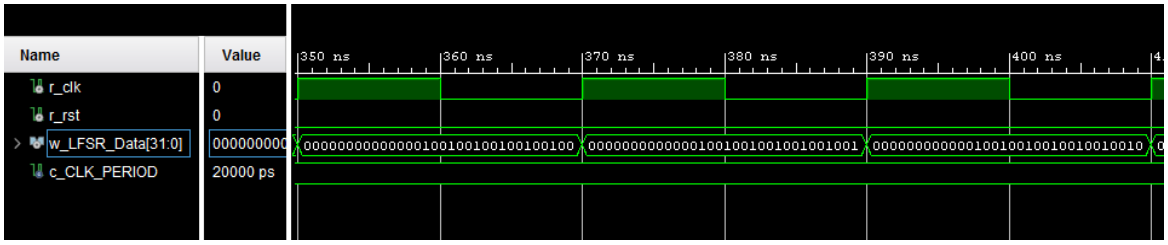



**Figura 0-3: Simulación LFSR**

En nuestro proyecto hemos realizado esta implementación, debido a que este sistema es muy utilizado en ambientes de desarrollo para la creación de pruebas, ya que su construcción es muy sencilla, basándose en circuitos electrónicos simples. Este proceso elimina la intervención del usuario a la hora de operar en las entradas.

Hay que destacar también que el tiempo en ejecutar un LFSR que produzca vectores de salida de N bits, siendo la N un número alto, producirá más retardo. Ya que realizamos pruebas de diferentes tamaños, el tiempo que tarda en ejecutarse el LFSR debe de ser el mismo, por ello para cada caso usaremos un LFSR de 8 bits.

De modo que en el caso en que necesitemos un numero de bits mayor que 8, concatenaremos ese vector sacado del LFSR, las veces que haga falta para cubrir el tamaño necesario de bits.

## *B. Generador de paridad parametrizable*

El generador de paridad parametrizable se trata de un sistema de comprobación de resultados, de tal manera que nos ayude a detectar los posibles errores en la transmisión del dato transmitido, y por consiguiente, la corrección de los fallos.
Estos errores son cambios indeseados en los bits entregados, es decir, de uno a cero o viceversa, producidos por componentes que funcionan en mal estado o por interferencias de agentes externos.
El método utilizado se basa en la transmisión del bit de paridad, este bit hace referencia al número de unos que existen en la palabra transmitida. Por tanto, se establecen dos casos:

- Paridad par: Esta condición ocurre cuando el número de unos de la palabra es par, el bit de signo será cero. Por tanto, si contamos todo el conjunto seguirá respetando la paridad inicial, en caso contrario el bit de paridad será uno mostrándonos que ha ocurrido un error.

- Paridad impar: En el caso en que el total de unos es impar, siendo el bit de signo cero. Como sucedía antes, si incluimos al conjunto el bit de paridad, seguirá siendo impar. Si sucediera el caso contrario obtendríamos un bit de paridad igual a uno.

La fórmula del bit de paridad es:

$$BitParidad = A_{N-1} \oplus A_{N-2} \oplus \dots \oplus A_2 \oplus A_1 \oplus A_0$$

Si lo planteamos para el caso de una entrada A de 5 bits, el esquema es el siguiente:

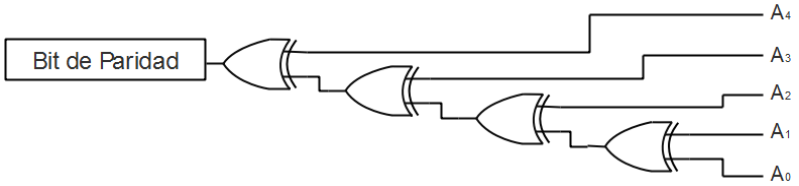



**Figura 0-4: Estructura generador de paridad**

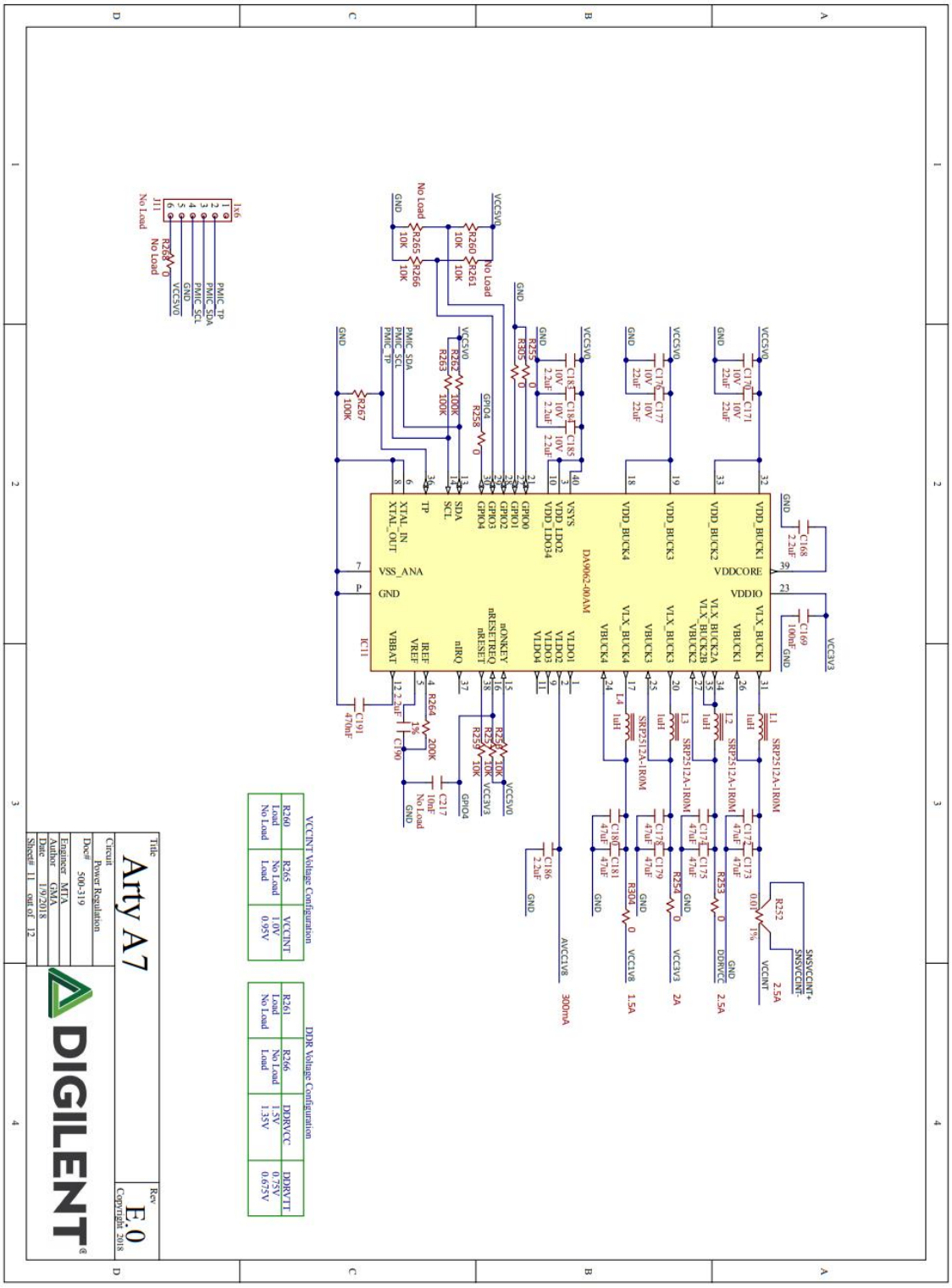
De modo que si establecemos un detector de paridad con convenio de paridad par:

| A | B | C | D | E | Bit de paridad |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| | | ... | | | ... |

**Tabla 0-1: Ejemplo generador de paridad**

El bit de paridad nos muestra que si es igual a uno, ha ocurrido un error en la transmisión, es decir, para el convenio que hemos propuesto el número de unos debía de ser par, pero en la secuencia final no ha sido así

## C. Esquemático de la Arty A7 35T (Sección: Regulación de Potencia)



## *D. Ejecución del comando TCL*

El proceso para la ejecución del script, una vez formado el proyecto, se debe comenzar a partir de la manera siguiente:

1. Ejecutar Run *Synthesis*, Run *Implementation* y Generate *Bitstream*.

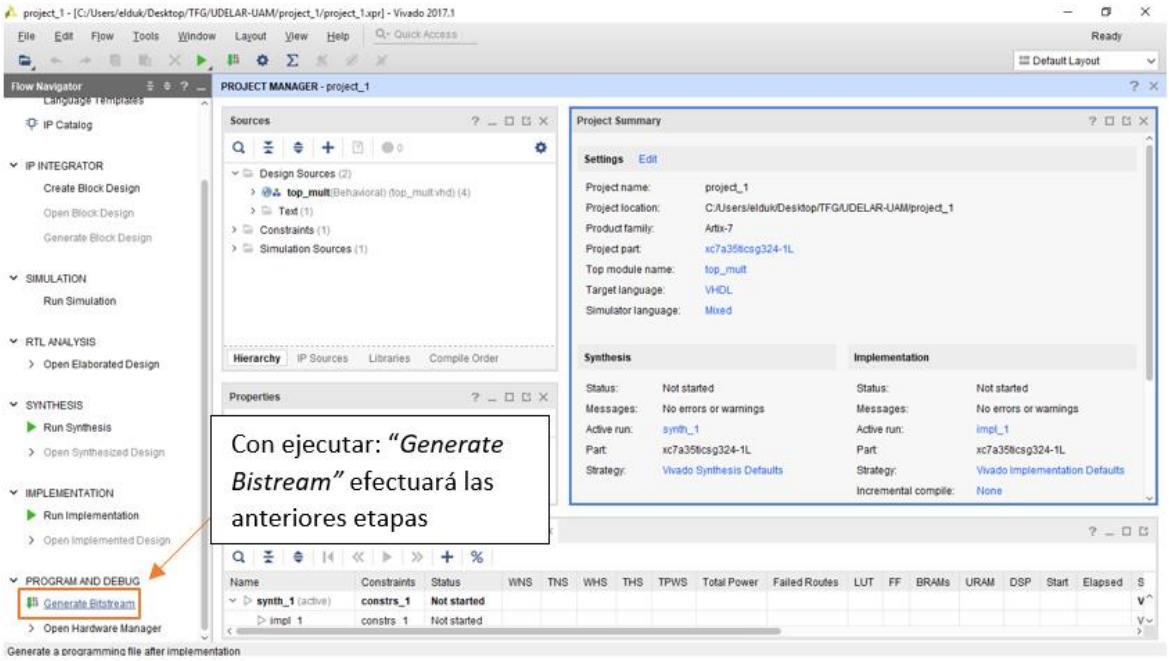



**Figura 0-5: Ejecución**

2. Con la Arty conectada, seleccionar: Open Target →Auto Connect.

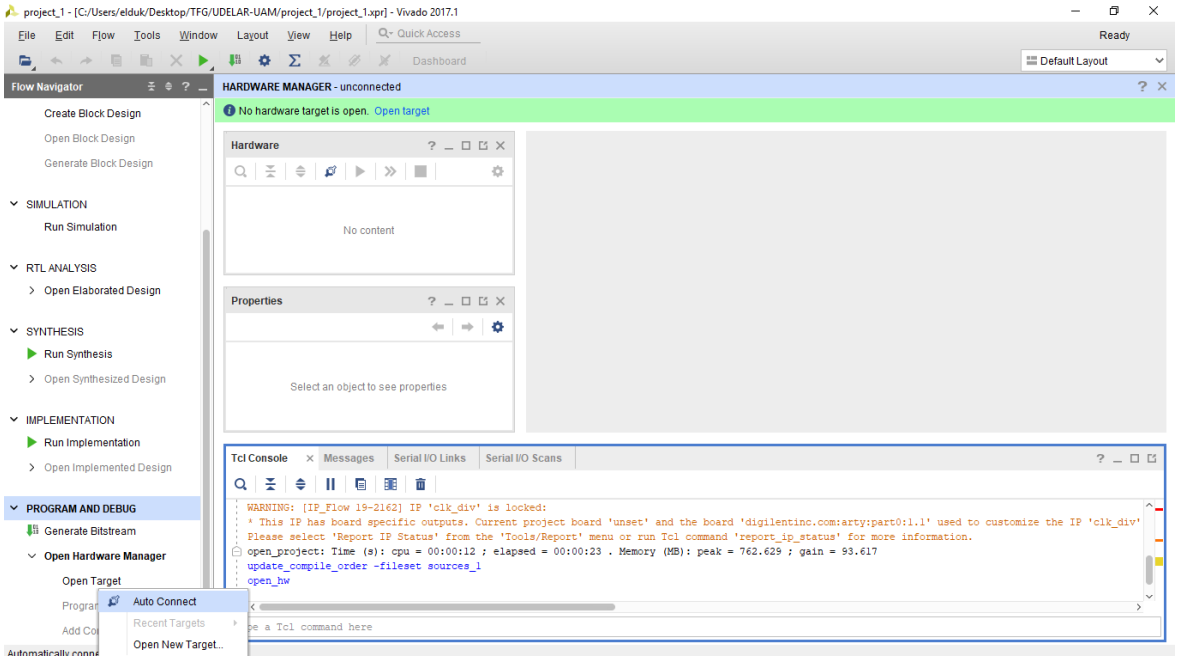



**Figura 0-6: Auto Connect**

3. El siguiente paso seleccionar: Program Device → xc7a35t_0 y seleccionar el "archivo".bit generado.

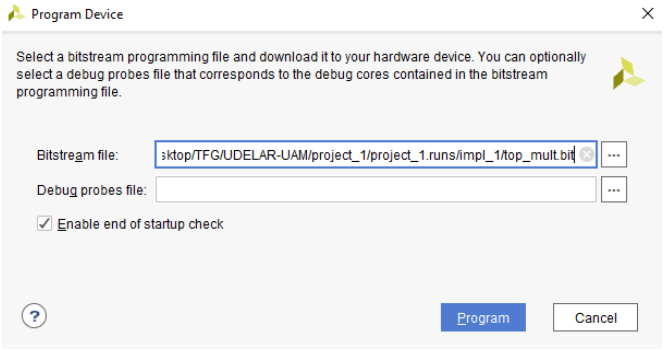



**Figura 0-7: "Archivo".bit generado**

4. Seleccionamos en la pestaña "Tools" → Run Tcl Script.

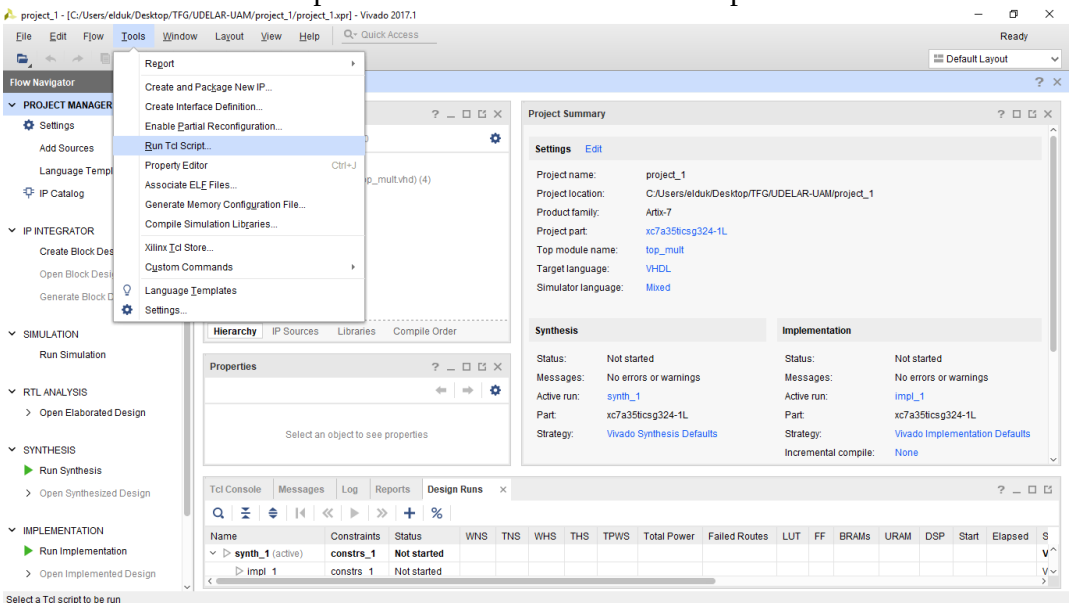



**Figura 0-8: Run TCL Script**

5. Buscamos el archivo "ug480_setup.tcl" en el directorio y le damos aceptar.

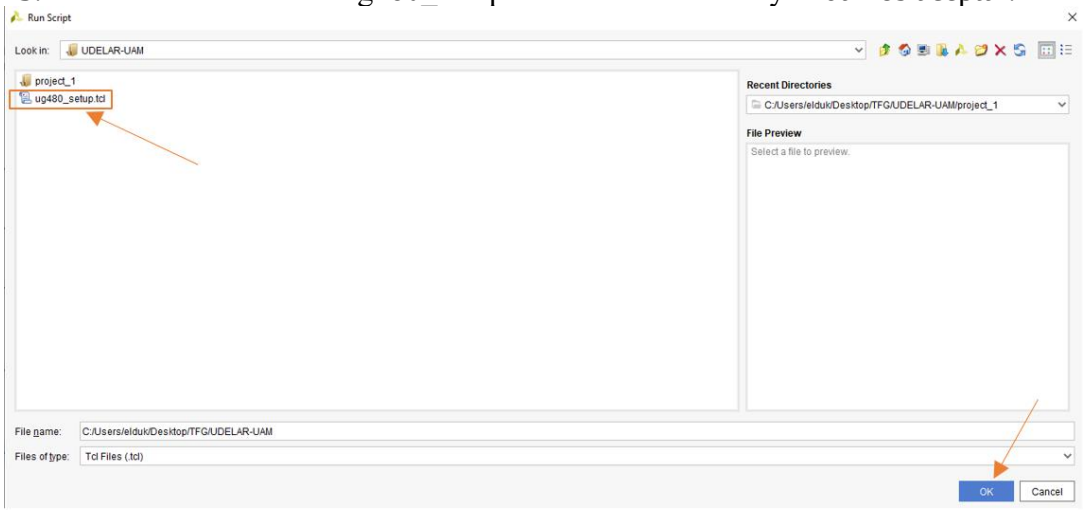



**Figura 0-9: Seleccionar comando "ug480_setup.tcl"**

6. Ejecutar "ug480_hw_read_aux_ch_10" en la consola de comando “Tcl Console”.

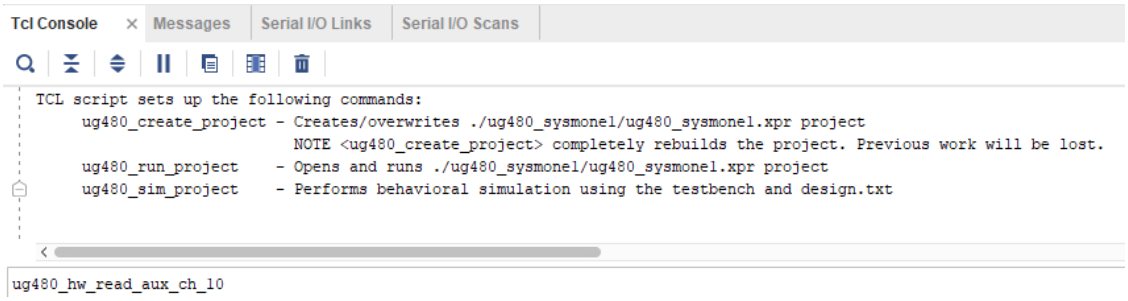



**Figura 0-10: Ejecución comando TCL**

Al efectuar el comando, se muestra por pantalla la tensión estimada para 100 valores

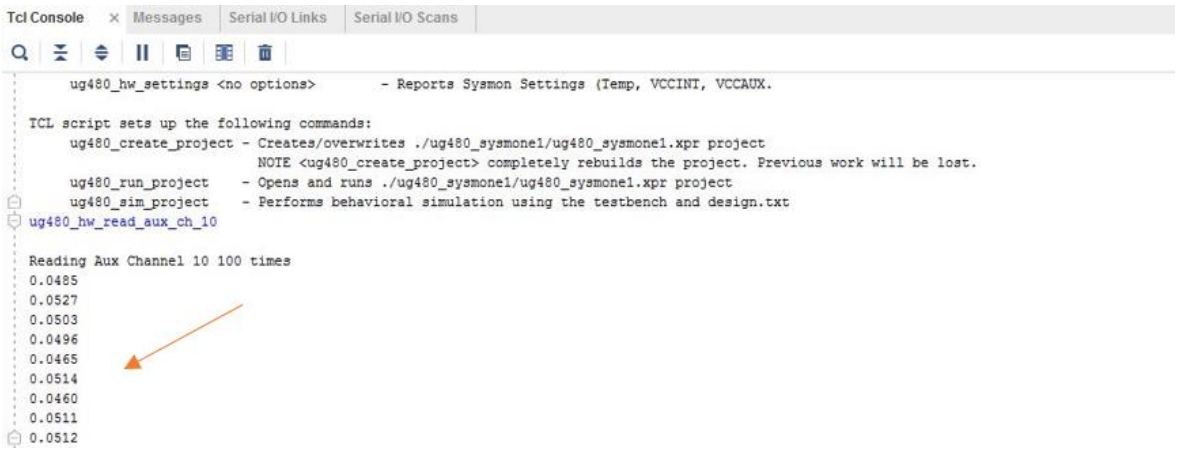



**Figura 0-11: Representación muestra de tensiones estimadas**

# 8 Apéndice

## 8.1 Código “TestBench” del LFSR

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity LFSR_N_bits_TB is
end entity LFSR_N_bits_TB;

architecture behave of LFSR_N_bits_TB is
  constant c_CLK_PERIOD : time := 20 ns;  -- 25 MHz
  signal r_clk : std_logic := '0';
  signal r_rst : std_logic := '0';
  signal w_LFSR_Data : std_logic_vector(7 downto 0) :=(others => '0');
begin
    r_Clk <= not r_Clk after c_CLK_PERIOD/2;
    r_rst<= '1', '0' after c_CLK_PERIOD/4;
  lfsr : entity work.lfsr_8_bits
    port map (
        clk  => r_Clk,
        rst  => r_rst,
        ce   => '1',
        lfsr_out => w_LFSR_Data
      );
end architecture behave;
```

## 8.2 Código del Multiplicador “Array Ripple”

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity multiplicador_Array_ripplecarry is
generic(
    N: integer:=8
);
port (
  A: in std_logic_vector (N-1 downto 0);
  B: in std_logic_vector (N-1 downto 0);
  P: out std_logic_vector (2*N-1 downto 0)
  );
end multiplicador_Array_ripplecarry;

architecture Behavioral of multiplicador_Array_ripplecarry is

component sumador_Ripplecarry_mux is
port (
  A: in std_logic_Vector (N-1 downto 0); -- Multiplicando
  B: in std_logic_Vector (N-1 downto 0); -- Multiplicador
  Ci: in std_logic;
  S: out std_logic_Vector (N-1 downto 0);
  Cout: out std_logic
  );
end component;
```

```
signal m_do : std_logic_vector (N-1 downto 0) := (others=>'0');
signal m_dor : std_logic_vector (N-1 downto 0) := (others=>'0');
signal prod_inter : std_logic_vector (N-1 downto 0) := (others=>'0');

type matriz is array (N downto 1) of std_logic_vector (N downto 0);
signal suma_acu: matriz := (others => (others => '0'));

begin

    m_do  <= A;
    m_dor <= B;

    -- Determinamos el primer número para la suma acumulada que
dependiendo del primer digito del multiplicador
    -- será 0 o el multiplicando
    suma_acu(1)(N-1 downto 0) <= m_do when m_dor(0)='1' else (others
=>'0');



    Multiplicador: for i in 1 to N-1 generate
        Suma_Acumulada: sumador_Ripplecarry_mux
             port map (
                A    => suma_acu(i) (N downto 1), -- Operando que lleva
la suma acumulada
                B    => m_do,                 -- Multiplicando
                Ci   => m_dor(i),              -- Bit que contiene el dato
del multiplicador para operar
                Cout => suma_acu(i+1)(N),          -- Acarreo final
obtenido colocado en la ultima columna de la suma acumulada final
                S    => suma_acu(i+1)(N-1 downto 0)-- Suma acumulada
            );

        prod_inter(i) <= suma_acu(i+1)(0); -- Se toma los valores de la
ultima columna, ya que a la siguiente etapa se pierde
    end generate Multiplicador;

    P(2*N-1 downto N)<= suma_acu (N)(N downto 1);
    P(N-1 downto 1)<=prod_inter(N-1 downto 1);
        -- Primer bit del producto
    P(0)<=suma_acu(1)(0);

end Behavioral;
```

### *8.3 Codigo "Ripple" más MUX*

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity sumador_Ripplecarry_mux is
generic (
    N: integer:=8
);
port (

    A: in std_logic_vector (N-1 downto 0);
    B: in std_logic_vector (N-1 downto 0);
    Ci: in std_logic;
```

```
    S: out std_logic_vector (N-1 downto 0);
    Cout: out std_logic
);

end sumador_Ripplecarry_mux;

architecture Behavioral of sumador_Ripplecarry_mux is

component full_adder
   port (
        A: in std_logic;
        B: in std_logic;
        Ci: in std_logic;
        S: out std_logic;
        Cout: out std_logic
        );
   end component;


signal s_c: std_logic_vector (N downto 0) := (others=>'0');
signal s_and: std_logic_vector (N-1 downto 0):= (others=>'0');
signal s_sum: std_logic_vector (N-1 downto 0):= (others=>'0');

    begin

s_and <= B when Ci='1' else (others=>'0'); --Si Ci = 1, s_and es el mismo
valor que el multiplicando, MUX
s_c(0)<='0';

    Sumador_Ripple_Carry: for i in 0 to N-1 generate
         Full_Adders: full_adder port map(
             A=>A(i),
             B=>s_and(i),
             Ci=>s_c(i),
             S=>s_sum(i),
             Cout=>s_c(i+1)
         );
    end generate Sumador_Ripple_Carry;


Cout <= s_c(N);
S <= s_sum;
end Behavioral;
```

## *8.4 Codigo Multiplicador "Array Look-Ahead"*

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity multiplicador_Array_lookahead is
generic(
    N: integer := 32
);
port (
  A: in std_logic_vector (N-1 downto 0);
  B: in std_logic_vector (N-1 downto 0);
```

```
  P: out std_logic_vector (2*N-1 downto 0)
  );
end multiplicador_Array_lookahead;

architecture Behavioral of multiplicador_Array_lookahead is

component sumador_Lookahead_mux is
port (
  A: in std_logic_Vector (N-1 downto 0); -- Multiplicando
  B: in std_logic_Vector (N-1 downto 0); -- Multiplicador
  Ci: in std_logic;
  S: out std_logic_Vector (N-1 downto 0);
  Cout: out std_logic
  );
end component;

signal m_do : std_logic_vector (N-1 downto 0) := (others=>'0');
signal m_dor : std_logic_vector (N-1 downto 0) := (others=>'0');
signal prod_inter : std_logic_vector (N-1 downto 0) := (others=>'0');

type matriz is array (N downto 1) of std_logic_vector (N downto 0);
signal suma_acu: matriz := (others => (others => '0'));

begin

    m_do <= A;
    m_dor<= B;

    -- Determinamos el primer numero para la suma acumulada que
dependiendo del primer digito del multiplicador
    -- será 0 o el multiplicando
    suma_acu(1)(N-1 downto 0) <= m_do when m_dor(0)='1' else (others
=>'0');

    Multiplicador: for i in 1 to N-1 generate
        Suma_Acumulada: sumador_Lookahead_mux
             port map (
                A    => suma_acu(i) (N downto 1), -- Operando que lleva
la suma acumulada
                B    => m_do,                 -- Multiplicando
                Ci   => m_dor(i),              -- Bit que contiene el dato
del multiplicador para operar
                Cout => suma_acu(i+1)(N),          -- Acarreo final
obtenido colocado en la ultima columna de la suma acumulada final
                S    => suma_acu(i+1)(N-1 downto 0)-- Suma acumulada
            );

        prod_inter(i) <= suma_acu(i+1)(0); -- Se toma los valores de la
ultima columna, ya que a la siguiente etapa se pierde
    end generate Multiplicador;

    P(2*N-1 downto N)<= suma_acu (N)(N downto 1);
    P(N-1 downto 1)<=prod_inter(N-1 downto 1);
        -- Primer bit del producto
    P(0)<=suma_acu(1)(0);

end Behavioral;
```

## 8.5 Código del Look-Ahead más MUX

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity sumador_Lookahead_mux is
generic (
    N: integer:= 32
);
port (

    A: in std_logic_vector (N-1 downto 0);
    B: in std_logic_vector (N-1 downto 0);
    Ci: in std_logic;
    S: out std_logic_vector (N-1 downto 0);
    Cout: out std_logic
);

end sumador_Lookahead_mux;

architecture Behavioral of sumador_Lookahead_mux is

component full_adder
   port (
        A: in std_logic;
        B: in std_logic;
        Ci: in std_logic;
        S: out std_logic;
        Cout: out std_logic);
   end component;

signal s_and: std_logic_vector (N-1 downto 0);
signal s_sum: std_logic_vector (N-1 downto 0);

signal s_g : std_logic_vector(N-1 downto 0); -- Señal de generación
signal s_p : std_logic_vector(N-1 downto 0); -- Señal de propagacion
signal s_c : std_logic_vector(N downto 0);

    begin

s_and <= B when Ci='1' else (others=>'0'); --Si Ci = 1, es el mismo valor
que el multiplicando, MUX
s_c(0)<='0';
    Sumador_LookAhead: for i in 0 to N-1 generate
         Full_Adders_LookAhead: full_adder port map(
             A=>A(i),
             B=>s_and(i),
             Ci=>s_c(i),
             S=>s_sum(i),
             Cout=>open
         );

         s_g(i)   <= A(i) and s_and(i);
         s_p(i)   <= A(i) or s_and(i);
         s_c(i+1) <= s_g(i) or (s_p(i) and s_c(i)); -- Calculo del valor
del acarreo

    end generate Sumador_LookAhead;
```

```
Cout <= s_c(N);
S <= s_sum;
end Behavioral;
```